TECHNISCHE UNIVERSITÄT DRESDEN

Software Technology Group

Fakultät Informatik Institut für Software- und Multimediatechnik, Lehrstuhl für Softwaretechnologie

Diplomarbeit

EINFLUSS VON SOFTWAREARCHITEKTUR AUF DEN WERT EINES SOFTWARESYSTEMS

bearbeitet von

Hendrik Schön

geboren am 19. Juni 1990 in Dresden

Betreuer:

Prof. Dr. Frank J. Furrer

Hochschullehrer:

Prof. Dr. rer. nat. habil. Uwe Aßmann

Eingereicht am 27.06.2016

Diplom-Aufgabenstellung für Hendrik Schön

von

Dr. Frank J. Furrer / TU Dresden

V1.0 / 10. April 2015

Einfluss der Softwarearchitektur auf den Wert eines Softwaresystems

Software ist heute allgegenwärtig. Man findet sie in fast allen Produkten und Diensten der heutigen Gesellschaft. Die Abhängigkeit von Software ist nahezu total geworden. In den letzten zwei Jahrzehnten hat sich Software für die Unternehmen von Systembausteinen zu signifikanten *Investitionsgütern* entwickelt, welche die Chancen und Risiken der Unternehmen stark mitbestimmen. Viele Softwaresysteme haben eine lange Lebensdauer und müssen während dieser Zeit kontinuierlich neuen Anforderungen angepasst werden. Dabei sind zunehmende Anforderungen an Qualitätseigenschaften – wie Sicherheit, Verfügbarkeit, gesetzliche Vorschriften, usw. – nachweisbar zu erfüllen. Dies stellt sehr hohe Anforderungen an die Software selbst und an ihre Evolutionsstrategie. Kurz gefasst muss die Software *zukunftsfähig* sein, d.h. sie muss sowohl die heutigen Anforderungen optimal erfüllen, aber gleichzeitig ihre erfolgreiche Evolution für längere Zeit gewährleisten.

Der *Wert* des Investitionsgutes „Softwaresystem“ verlangt eine betriebswirtschaftliche Sichtweise: Sein Wert – sowohl als *operatives* System wie auch als langfristige *finanzielle Investition* – soll nicht nur erhalten, sondern kontinuierlich erhöht werden. Als Metriken für den Wert des Softwaresystems gelten sein *Geschäftsnutzen* [GeN] (= „Business Value“ in [1]) und seine optimale *Evolutionsfähigkeit* [EvF] (= „Agility“ in [1]), d.h. seine gute Anpassungsfähigkeit an zukünftige, heute noch unbekannte Anforderungen.

Die Grundlage eines zukunftsfähigen Softwaresystems ist seine *Architektur*, d.h. die grundlegende Organisation seiner Teile und ihrer Beziehungen untereinander und zur Umwelt. Über den Zusammenhang zwischen guter Architektur und Evolutionsfähigkeit ([2], [3]), resp. zwischen schlechter Architektur und den resultierenden Evolutionsproblemen ([4], [5]) existiert eine reiche, *empirische* Literatur.

Nur sehr wenige Literaturstellen versuchen, einen direkten, *kausalen Zusammenhang z*wischen Architektureigenschaften und dem Wert eines Softwaresystems (in dieser Arbeit: gemessen als *Evolutionsfähigkeit*) aufzuzeigen.

Ziel dieser Diplomarbeit ist, ein *Architekturprinzip* aus den grundlegenden Architekturprinzipien ([6], [7]) auszuwählen und den direkten, kausalen – wenn möglich quantitativen – Einfluss auf die Evolutionsfähigkeit herzuleiten. Damit wäre ein schöner *Beweis* (mehr als empirisch!) für den positiven Einfluss guter Softwarearchitektur auf den Wert eines Softwaresystems erbracht.

Diese Aufgabe ist nicht nur eine anspruchsvolle konzeptionelle Problemstellung, sondern hat auch eine direkte und wichtige *betriebswirtschaftliche* Konsequenz für die Praxis ([8]): Die *Gewissheit*, dass sich seriöse, konsequente und gute Softwarearchitektur positiv – als Werterhalt und Wertvermehrung des Softwaresystems – auszahlt.

Aufgabenstellung

Kontext

Software ist heute allgegenwärtig: Man findet sie in fast allen Produkten und Diensten der heutigen Gesellschaft. Die Abhängigkeit von Software ist nahezu total geworden. Viele Softwaresysteme sind heute sehr langlebig, d.h. sie müssen zukunftsfähig gebaut und unterhalten werden. Ihr Wert hängt von ihrer Zukunftsfähigkeit ab.

Zukunftsfähige Softwaresysteme sind eine Folge ihrer Architektur. Eine gute Architektur bestimmt deren Qualitätseigenschaften und damit ihren Wert als operationelle Systeme und als Investitionsgut.

Gute Architektur ist eine Folge von kontinuierlicher Umsetzung bewährter Architekturprinzipien. Nur die konsequente Beachtung von Architekturprinzipien führt zu langlebigen, zukunftsfähigen Softwaresystemen.

Diese Diplomaufgabe hat zum Ziel, den kausalen – wenn möglich quantitativen – Zusammenhang zwischen Architekturprinzipientreue und dem Wert eines Softwaresystems nachzuweisen.

Zielsetzung und angestrebte Arbeitsergebnisse

- Literaturstudium und systematische Darstellung relevanter Arbeiten.
- Erarbeitung theoretischer Möglichkeiten für den Nachweis des kausalen Zusammenhangs zwischen dem Wert (in der Form der Evolutionsfähigkeit) eines Softwaresystems und der Konformität zu einzelnen Architekturprinzipien.
- Auswahl des Architekturprinzips „Datenredundanz“ und Modellierung dessen Einflusses auf den Wert – d.h. auf die Evolutionsfähigkeit – des Softwaresystems.
- Herleitung von quantitativen Formeln für die Abhängigkeit des Wertes des Softwaresystems vom Konformitätsgrad zum Architekturprinzip „Datenredundanz“.

Danksagung

Besonderer Dank geht an *Prof. Dr. Frank J. Furrer*, der mich bei diesem Thema und dem Verständnis der Zusammenhänge unterstützt und die Fachkompetenz eingebracht hat. Durch unsere Zusammenarbeit in diesem recht neuartigen Forschungsfeld konnte diese Arbeit viel an Wert gewinnen und macht sie hoffentlich in der heutigen Welt der Softwarearchitektur noch relevanter.

Weiterhin bedanke ich mich bei *Prof. Dr. Uwe Aßmann*, da er mir die Möglichkeit geboten hat, diese Diplomarbeit an seinem Lehrstuhl für Softwaretechnologie zu erarbeiten und die dafür notwendigen Voraussetzungen geschaffen hat.

Großer Dank gilt auch all jenen (ganz besonders *Tim Hackel*), die mir inhaltlich weitergeholfen haben, sei es durch die Ideendiskussion oder bei mir fachfremden Themen, bei welchen ich selbst entweder kein Fachwissen besitze oder weitere Meinungen zu Lösungen einholen musste, um anschließend die Lösungen zu diskutieren.

Ich bedanke mich bei allen Korrekturlesern, welche diese Arbeit erst lesbar gemacht haben. Auf knapp 190 Seiten finden sich immer einige Fehler, welche einem selbst nicht mehr auffallen, weil man irgendwann den eigenen Blick dafür verliert.

Zu guter Letzt bedanke ich mich bei allen, welche mir in den letzten fünf Jahren des Studiums immer geholfen und mich unterstützt haben, das Studium zu vollenden und die Motivation beizubehalten. Danke für all die interessanten Arbeiten und Projekte, die wir zusammen bearbeiteten und für all das, was wir unternommen haben und über das Studium hinaus ging.

Vorwort

Software ist in den letzten Jahrzehnten auf vielen Ebenen weiterentwickelt worden. Software hat sich an jeweils aktuelle Gegebenheiten angepasst und ist ein Ankerpunkt im heutigen digitalen Zeitalter. Wurde in den Anfängen der Softwareentwicklung noch in simplen Schemata gedacht, kommt heute bei groß angelegter Software ein Produktionszyklus zum Vorschein, der sich von dem Anforderungsmanagement bis hin zu den Testumgebungen und den Wartungsmechanismen erstreckt. In all diesen Phasen sind neue Berufszweige entstanden und neue Forschungsfelder taten sich auf.

In dieser Diplomarbeit geht es nur um einen bestimmten Ausschnitt aus diesem Zyklus, der Definition der Architektur von Software. Dabei ist Softwarearchitektur ein zentraler Baustein in der Entwicklungskette von Software. Besonders bei Großprojekten kann dieser Teil durchaus einen der kritischen Punkte darstellen. In dieser Arbeit, geschrieben an der Technischen Universität Dresden an der Fakultät Informatik am Lehrstuhl für Softwaretechnologie, soll dieser wichtige Punkt der Architektur veranschaulicht werden, und das in einer Weise, die vorher noch nicht betrachtet wurde.

Die Thematik der Softwarearchitektur wurde von mir selbst ausgewählt. Gerade in dieser Zeit hielt *Prof. Dr. Frank J. Furrer* an der TU Dresden Vorlesungen zu dem Thema Softwarearchitektur für zukunftsfähige Systeme. Eine Anfrage zu einer Diplomarbeit führte zu dem nun vorliegenden Ergebnis mit ihm als Betreuer der Arbeit. Die Motivation dazu, ein solch theoretisches Thema mit relativ geringem Programmieranteil zu wählen, liegt in meinem Interesse an Softwarearchitektur und -Design begründet. Ich halte beides für wichtige Punkte, welche gute Software ausmachen und die Strukturen vorgeben, in denen große neue Software-Ideen realisiert werden können. Und zu guter Architektur gehören eine Menge Einflüsse, unter anderem die Umsetzung von Richtlinien im System, den *Architecture Principles*, was der Ausgangspunkt der Themenfindung war. Diese Arbeit beleuchtet eine neuartige Thematik: einen formalen Nachweis für gute Architektur. Es ist mir bewusst, dass es dazu kaum mehr als empirische Literatur gibt, aber das gab den Anreiz. Im Ergebnis stellt die Arbeit vielleicht kein „Werkzeug“ für die Praxis dar. Aber der Einfluss auf das Thema der Softwarearchitektur ist hoch und eventuell lässt sich nun, wo der Wert von Softwarearchitektur dargelegt werden kann, der gesicherte Einfluss von Architektur in bessere Software umsetzen.

Dabei ist die Arbeit wie folgt strukturiert: zunächst werden im ersten Kapitel die Motivation und Ziele dieser Arbeit beleuchtet, sowie erste Voraussetzungen aufgezeigt. Im zweiten Kapitel wird es sich um die Grundlagen von Software und deren Architektur handeln, welche benötigt werden, um das Thema zu verstehen. Im dritten Kapitel wird näher auf Methoden und Metriken eingegangen und genauer erläutert, welche Möglichkeiten wofür in Betracht kommen und wie man sie für die Ziele dieser Arbeit nutzen kann. Das vierte Kapitel ist der Redundanz gewidmet, welche einen Kernaspekt der Arbeit darstellt. Im fünften Kapitel beginnt der eigentliche Nachweis der Arbeit. Dabei werden die in den vorherigen Kapiteln aufbereiteten Grundlagen genutzt, um das Ziel der Arbeit zu vollenden. Das sechste Kapitel beschreibt ein Simulationsprogramm, mit welchem die Ergebnisse aus dem fünften Kapitel nachvollzogen und Testszenarien ausgewertet werden. Im siebten und letzten Kapitel werden alle gefundenen Ergebnisse aufbereitet und der Nachweis sowie dessen Auswirkungen rekapituliert.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Das Investitionsgut „Software“ . . . 1
- 1.2 Motivation für den Nachweis von Werten der Architektur . . . 2
- 1.3 Zielsetzung . . . 3
- 1.4 Verwandte Arbeiten und State of the Art . . . 4
- 1.5 Begriffe und Erläuterungen . . . 8

2 Grundlagen von Softwaresystemen **11**
- 2.1 Eigenschaften . . . 11
  - 2.1.1 Ausmaße und Dimensionen . . . 11
  - 2.1.2 Die Kosten eines Systems . . . 13
  - 2.1.3 Der Wert eines Systems . . . 15
- 2.2 Grundlagen von Softwarearchitektur . . . 17
  - 2.2.1 Architektur in Softwaresystemen . . . 18
  - 2.2.2 Einfluss und Auswirkung von Architektur . . . 19
  - 2.2.3 Architekturprinzipien . . . 20
- 2.3 Fehler durch Architektur und Management . . . 23
- 2.4 Zukunftsfähige Softwaresysteme . . . 27
  - 2.4.1 Langlebige Software . . . 27
  - 2.4.2 Zukunftsfähige Architektur: Agility, Resilience und der Business Value . . . 29
  - 2.4.3 Managed Evolution . . . 32

3 Methodik und Metriken **35**
- 3.1 Methodik-Entwicklung . . . 35
  - 3.1.1 Darstellbarkeit . . . 37
  - 3.1.2 Messbarkeit . . . 38
- 3.2 Softwaresysteme als Graphen . . . 39
  - 3.2.1 Eigenschaften . . . 41
  - 3.2.2 Berechnungen und Verhalten . . . 43
- 3.3 Graphen als Werkzeug für Architecture Principles . . . 43
- 3.4 Metriken in Softwaresystemen . . . 44
  - 3.4.1 Software Produkt Attribute . . . 45
  - 3.4.2 Messen von Metriken im System . . . 46
  - 3.4.3 Bauen von Metriken . . . 47

4 Redundanz **51**
- 4.1 Redundanz im System . . . 51
  - 4.1.1 Managed Redundancy . . . 54
  - 4.1.2 Unmanaged Redundancy . . . 54
- 4.2 Auswirkungen und Folgen . . . 55

5 Der Nachweis **59**
5.1 Problematik und Methodik . . . 59
5.2 I: Das Graphmodell . . . 62
5.2.1 Erstellung des Graphen . . . 64
5.2.2 Eigenschaften . . . 66
5.2.3 Evolution . . . 75
5.3 II: Die Metriken . . . 76
5.3.1 Graph-Metriken . . . 78
5.3.2 Wert-Metriken . . . 92
5.4 III: Der Wert des Systems . . . 111
5.4.1 Ermittlungen der Parameter . . . 112
5.4.2 Die Wert-Formel . . . 115
5.5 Dynamisches Verhalten . . . 120
5.6 Fehleranalyse und Toleranzen . . . 123

6 System Simulation **129**
6.1 Intention und Korrektheit . . . 129
6.2 Das Simulation Tool . . . 130
6.2.1 Anforderungen und Spezifizierungen . . . 130
6.2.2 Entwicklung und Funktionsweise . . . 132
6.2.3 Darstellung des Nachweises . . . 137
6.3 Auswertung der Simulation . . . 140
6.3.1 Bezug zur Simulation . . . 141
6.3.2 Bezug zur Theorie . . . 143

7 Schlussfolgerung und Abschluss **151**
7.1 Kritik und offene Fragen . . . 151
7.2 Weiterführende Möglichkeiten . . . 155
7.3 Zusammenfassung und Einschätzung . . . 158

A Appendix **i**
A.1 Berechnung der Metriken am Beispielsystem . . . i
A.2 Belegung der Parameter des Programms für das Testszenario . . . v

B Inhalt elektronischer Datenträger (DVD) **vii**

1 Einleitung

1.1 Das Investitionsgut „Software“

Software ist heutzutage überall anzutreffen. Man findet sie in fast allen Produkten und Diensten der heutigen Gesellschaft. Sie bildet die Grundlage moderner Technologie, von Weltraumforschung, über den Finanzsektor bis hin in den sozialen und medizinischen Bereich. Im Laufe der letzten Jahrzehnte hat sich Software damit vom Werkzeug zu einem wesentlichen Bestandteil der heutigen Wirtschaft und Forschung und damit einem signifikanten Investitionsgut entwickelt, welches Chancen und Risiken eines Unternehmens erheblich beeinflusst. Durch diese zunehmende Abhängigkeit von Softwaresystemen entsteht auch der Zwang, Softwaresysteme über eine lange Zeitperiode zu nutzen und zu erweitern, gleichzeitig aber auch dieses System stetig zu verbessern. Dabei sind zunehmend diverse Qualitätsattribute von Software bestimmend, wie zum Beispiel Sicherheit, Verfügbarkeit, Performance und gesetzliche Vorschriften, welche vertraglich vereinbart werden und damit auch nachweisbar erfüllt werden müssen. Dies stellt sehr hohe Anforderungen an die Konstruktion und Evolution eines Softwaresystems. Um diese Anforderungen erfüllen zu können und gleichzeitig wirtschaftliche Ansprüche zu bedienen, erfordert es eine *zukunftsfähige* Software, was bedeutet, dass die sowohl für heutige Verhältnisse, aber auch für zukünftige Erweiterungen gebaut wurde und der Erfolg gewährleistet ist. Damit ist Software gleichzusetzen mit anderen Produkten und Investitionen eines gewinnorientierten Unternehmens, welches mit einer endlichen Menge an Budget das Produkt auf dessen derzeitigen und zukünftigen Nutzen hin evaluieren muss.

Dass sich Investitionen in die heutigen und zukünftigen Softwaresysteme lohnen, belegen derzeitige Statistiken für Ausgaben von Firmen im IT-Sektor [41] [66]. Immerhin sind solche Systeme ein wichtiger Stützpfeiler für das Business geworden, welcher nicht wegzudenken wäre. Dabei dienen die Softwareprodukte nicht nur der Unterstützung, sondern werden aktiv vorangetrieben, um Vorteile gegenüber Konkurrenten zu erhalten. Zwei Beispiele für solche extremen Investitionen in die Softwaresysteme sind das *High Frequency Trading* [10] [45] und das Handeln und Verwalten von *Bitcoins* [30] [40]. Bei beiden Szenarien dient das zugrundeliegende System nicht mehr nur dem unterstützen der Mitarbeiter, sondern das System selbst ist der Hauptankerpunkt im Business und damit der Mehrwert erzeugende Faktor im Unternehmen. Aber auch wenn das System selbst nicht den Hauptankerpunkt der Unternehmensidee darstellt, können gut gepflegte und konkurrenzfähige Systeme eine bedeutende Rolle im umkämpften Markt darstellen, wie es beispielsweise bei der Entwicklung autonomer Drohnen oder bei neuen Weltraumoperationen der Fall ist. Hierbei ist das Softwaresystem nicht unmittelbar die Hauptinvestition, jedoch ein Kernaspekt, worauf sich das eigentliche Unternehmensziel stützt. Aber auch im Handwerk wie dem klassischen Autobau sind die neuen Features und Innovationen bei neuen Automodellen zum größten Teil neue und verbesserte Softwarelösungen [19] [9] angefangen vom Komfortfeatures wie Unterhaltungssysteme und Assistenten bis hin zu Sicherheitsfeatures durch Softwareunterstützung bei der Analyse der Fahrsysteme, der Umgebung sowie des Fahrverhaltens. Die Entwicklungen und Tendenzen hin zu einem autonomen Fahrzeug ist dabei nicht weniger abhängig von gut strukturierten Softwaresystemen. Eine fehlerhafte Einschätzung der Investi-

tionen in ein solches System hätte schwerwiegende Folgen für den zukünftigen Verlauf der Firma selbst sowie im Vergleich zu den existierenden Konkurrenten.

Ein Softwaresystem kann und muss daher mittlerweile betriebswirtschaftlich als „Investitionsgut“ angesehen werden. Der Wert eines solchen Softwaresystems, sowohl als Nutz-System als auch als langfristige Investition in die Zukunft, muss nicht nur auf dem gleichen Level gehalten werden, sondern kontinuierlich erhöht werden, um zum einen den Eigenerhalt des Unternehmens zu gewährleisten, und zum anderen die eigene Konkurrenzfähigkeit auszubauen. Als Metrik (Vergleichswert) für diese Zukunftsfähigkeit eines System gelten neben dessen Geschäftsnutzen („Business Value“) auch die Evolutionsfähigkeit („Agility“) und Widerstandskraft („Resilience“) des Systems. Aber gerade eine Evolutionsfähigkeit ist die wichtige Eigenschaft, dem System eine notwendige und geeignete Evolution zukommen zu lassen, um sich an zukünftige, heute noch unbekannte Anforderungen schnell und effektiv anzupassen.

Die Grundlage jeden guten Systems liegt in dessen Architektur, dem beschriebenen Zusammenhang von allen Teilen und deren Beziehungen untereinander, dem inneren Aufbau und der Organisation der Teile und Abläufe im jeweiligen Kontext des Systems. Der Zusammenhang zwischen guter Architektur und resultierender guter Evolutionsfähigkeit und Zukunftsfähigkeit eines Systems ist generell in den Meinungen vertreten. Jedoch liegen dazu nur wenige Literaturstellen vor, welche die Thematik tatsächlich mehr als nur empirisch aufzeigen. Auch für das umgekehrte Szenario, wenn ungenügende Architektur in einer relativ ungünstigen Zukunftsfähigkeit resultiert, ist kein direkter kausaler Zusammenhang gegeben. Damit steht der direkte kausale Zusammenhang zwischen Architektureigenschaften und dem Wert eines Softwaresystems (in dieser Arbeit gemessen als Wert für Evolutionsfähigkeit) noch aus. Wäre hier ein mehr als empirischer Zusammenhang nachgewiesen, hätte dies vermutlich enorme Auswirkungen auf betriebswirtschaftlicher Ebene in der Praxis. Das Resultat wäre ein Statement, welches als Grundlage für Entscheidungen dienen kann, da ein nachgewiesenes Statement glaubwürdiger erscheint, als einfache Annahmen.

1.2 Motivation für den Nachweis von Werten der Architektur

Dass Architektur ein durchaus gesonderter und ebenso wichtiger Baustein in der Kette der Entwicklung und Evolution eines Softwaresystems ist, das wird derzeit kaum noch jemand bestreiten wollen. Die Architektur des Systems, und damit die Grundlage des Ganzen und als Rahmen für die Entwicklung, hat damit direkt Auswirkungen auf das Endresultat des Software-Produkts. Jedoch ist es schwierig, mit allgemeinen Redewendungen und grob gefassten Regeln auf wirtschaftlicher Ebene zu hantieren. Es existiert bislang nach bestem Wissen und Gewissen des Autors keinerlei Literatur, welche versucht dieses Thema kausal und quantitativ anzugehen. Daraus ergibt sich die Motivation für einen Nachweis, dass sich die Investition in eine gute Architektur auch wirklich auszahlt, ebenso wie sich die Vernachlässigung dergleichen irgendwann negativ auf die Bilanz erstreckt. Damit hätte man am Ende einen handfesten Zusammenhang erwiesen, wovon mehrere Bereiche in der Entwicklungskette einer Software profitieren könnten.

Um nun eine Arbeit über einen Nachweis, oder eher einen Beweis über sich lohnende Softwarearchitektur, zu schreiben, ist es notwendig, sich die Sinnhaftigkeit eines Beweises vorzuhalten. Ein Beweis dient dabei nicht vorrangig als Werkzeug für praxisorientierte Methoden, sondern soll Sicherheit und Aufschluss geben. Die Vorgehensweisen, welche den Beweis ausmachen, sind dabei vielfältig. Einige schöne Beispiele und Kategorisierungen findet man in [61] [62], welche bei Interesse auch jeweils weiterführende Referenzen aufweisen.

Die Natur eines Nachweises (oder Beweises) ist im Allgemeinen nicht die Erstellung eines

Werkzeugs für ein spezielles Problem, sondern vielmehr die Sicherheit, welche durch einen kausalen Nachweis entsteht. Auch wenn es in dieser Arbeit nicht um die mathematischen Beweise sondern eher um abstraktere logische Schlussfolgerungen geht, sei hier doch einmal ein Satz aus dem Sach-Roman „Fermats letzter Satz“ von Singh notiert:
„Mathematische Beweise beruhen auf diesem logischen Verfahren, und einmal gelungen, sind sie wahr bis an das Ende der Zeit.“ [49], S. 45.
Es ist jedoch schwer, einen Nachweis von Softwarearchitektur im mathematischen Sinne zu vollführen, da die dafür vorher notwendigen formalen Axiome und Theoreme einfach nicht gegeben sind. Dafür lässt sich der Nachweis in dieser Arbeit mehr mit den Beweisen aus den Naturwissenschaften vergleichen, wofür Singh ebenfalls eine populärwissenschaftliche Erklärung parat hat:
„In der Naturwissenschaft wird eine Hypothese aufgestellt, um ein bestimmtes Phänomen zu erklären. Stimmen die Beobachtungen gut mit der Hypothese überein, gilt dies als Beleg zu ihren Gunsten. Die Hypothese sollte zudem [...] auch die Ergebnisse anderer Phänomene voraussagen können. [...] Schließlich kann die schiere Menge der Belege so beeindruckend werden, dass die Hypothese als naturwissenschaftliche Theorie anerkannt wird.“ [49], S. 45.
In diese Kategorie fällt auch der Beweis von Eigenschaften der Softwarearchitektur, auch wenn dies nicht direkt eine Naturwissenschaft ist. Um Fehlannahmen zuvor zu kommen, gibt es natürlich auch mathematische Beweise für Software und deren verbaute Algorithmen, zum Beispiel für eine Endlichkeit der Laufzeit eines Algorithmus oder für die Fehlerfreiheit in einem Programmablauf. Insgesamt ist jedoch das Spektrum an Beweisbarkeit bei Software relativ klein und der entsprechende Aufwand sehr hoch. Die Softwarearchitektur oder das Softwaredesign jedoch sind abstrakte Modelle, welche auf unzähligen verschiedenen Wegen umgesetzt werden können. Der Nachweis, dass eine Verletzung von Softwarearchitektur etwas Bestimmtes zur Folge hat, kann also kein rein mathematischer Beweis sein. Dafür bedeutet die Kategorie eines naturwissenschaftlichen Beweises keineswegs etwas Schlechtes, da auch diese Beweise eine Art der *höchstwahrscheinlichen* Sicherheit schaffen und für aktuelle und zukünftige Phänomene eine Erklärung und im besten Falle eine quantitative sowie qualitative Analyse liefern.

1.3 Zielsetzung

Ziel dieser Diplomarbeit war es, ein Architekturprinzip aus den grundlegenden Architekturprinzipien auszuwählen und den direkten, kausalen – und wenn möglich quantitativen – Einfluss auf die Evolutionsfähigkeit (den „Wert“) herzuleiten. Damit wäre mehr als nur empirisch nachgewiesen, dass sich gute Softwarearchitektur auf längere Sicht auch positiv auf den Wert des Softwaresystems auswirkt. Am Ende stünde dann die Gewissheit, dass sich Investition in gute Softwarearchitektur auszahlt.

Für den konkreten Fall dieser Arbeit wurde das Prinzip der (Daten-)Redundanz ausgewählt, weil dieses sowohl eine hohe Relevanz als auch einen hohen Praxisbezug mitbringt. Was genau das Prinzip der (Daten-)Redundanz besagt, wie man dies umsetzt und was dies eigentlich für den Wert des Systems im Sinne der Evolutionsfähigkeit aussagt, das sollen die folgenden Kapitel aufweisen.

Es wird aufgezeigt, wie genau man die Redundanz wahrnimmt, welche Folgen dies im System verursacht und wie man solche Fälle behandelt. Zudem wird erläutert, wie man aus der Redundanz-Eigenschaft eines konkreten Systems über Modellierung und Berechnungen den Wert des Gesamtsystems nachweisen kann. Dabei sind viele Themen aus Eigenarbeit entstanden, da hier die existierende Literatur nur teilweise vorhanden ist.

Der Weg über die Kausalität führt während der Arbeit über die Konstruktion von Modellen für die Eigenschaften eines Systems, über die Nutzung bekannter sowie Erschaffungen neuer Metriken hin zu der Definition des Wertes des Systems, bezogen auf die Redundanz im System. Die eigentliche Auswahl des Modells und die Auswahl der Kriterien für die Metriken sollen ebenfalls umfangreich dargestellt werden. Dabei ist hier anzumerken, dass diese Arbeit weder einen Beweis im mathematischen Sinne darstellt, noch unmittelbar auf die Praxis anwendbar ist. Jedoch wird diese Arbeit den kausalen Zusammenhang im Design-technischen Sinne verwirklichen und den Weg für weitere Arbeiten zu diesem Thema vorbereiten. Am Ende steht der kausale und quantitative Nachweis, dass sich gute Architektur als Mehrwert des Systems auszahlt, und das ist für sich allein gestellt derzeit bereits ein großer Fortschritt.

1.4 Verwandte Arbeiten und State of the Art

Für das oben genannte Thema gibt es zum derzeitigen Stand nicht viele Vorarbeiten, welche sich mit dem kompletten Ablauf beschäftigen. Jedoch kann man einzelne Arbeiten finden, welche Teile dieser Diplomarbeit ebenfalls betrachten, wenn auch aus leicht abweichenden Blickrichtungen.

Ein aktueller Stand der Technik ist hier nur bedingt gegeben, da es, wie bereits erwähnt, bislang kaum Arbeiten über den direkten Zusammenhang gibt und eher nur empirische Studien und Modelle vorliegen. In der Fachwelt ist man sich jedoch einig, dass schlechte Architektur auf lange Sicht erheblichen Mehraufwand und Mehrkosten verursacht, was Entwicklungskosten und -Zeit betrifft. Ebenfalls wird der Unterhalt von großen und lange betriebenen Systemen immer aufwendiger, je länger die Architektur und deren Pflege außer Acht gelassen wird. Diese Statements sind bei Entwicklern und Managern zwar etabliert, jedoch gibt es bislang keinen Anhaltspunkt, woran man den tatsächlichen kausalen Zusammenhang zwischen Architekturfehlern und deren Kostenauswirkungen (in Bezug auf die Evolutionsfähigkeit) erkennen kann. Die meisten Modelle versuchen derzeit, die Fehler in der Architektur üben den Quellcode zu finden und eine empirische Einschätzung der zugehörigen Kosten vorzunehmen, was vielleicht für eine praktische Kosteneinschätzung geeignet ist, aber keinen wirklichen kausalen Nachweis darstellt. In diesem Abschnitt werden dennoch solche Arbeiten vorgestellt, welche sich mit Themen beschäftigen, die das Thema dieser Arbeit zumindest teilweise tangieren und dahingehend als sinnvolle Lektüre für Ideen und Anregungen für diese Arbeit betrachtet werden können.

Managing Technical Debt in Software-Reliant Systems Die erste verwandte Arbeit ist von Brown *et al.* [4]. In dieser versuchen die Autoren das Prinzip und die Auswirkungen der *Technical Debt* (dt. „Technische Schuld“) aufzuzeigen. Das Thema der *Technical Debt* wird später auch noch in dieser Diplomarbeit thematisiert, jedoch nicht so ausführlich behandelt wie in dem angesprochenen Paper von Brown *et al.* Der Artikel entstand aus einem Workshop, welcher zum Ziel hatte, das Phänomen der *Technical Debt* genauer zu identifizieren und offene Forschungsfelder zu dokumentieren.

Zuerst beschreiben die Autoren genauer, warum es zu dieser Schuld in Softwaresystemen kommen kann und welche Vor- sowie Nachteile von dieser eingebauten Schuld später im System bemerkbar sind. Insgesamt beziehen sie sich sehr oft auf den Begriff „Schuld“ und daraus resultierender Analogien. Treffend ziehen sie Vergleiche zu der benachbarten finanziellen Schuld, wovon die Abarbeitung und In-Kauf-nehmen der offenen Schulden hin zu der technischen Schuld abgleitet wird. Dabei referenzieren sie auf andere Arbeiten wie [18], wo genauer auf verschiedene (Entstehungs-)Arten von *Technical Debt* eingegangen wird, indem diese in eine Schnittmenge

der vier Kategorien „Rücksichtslos“, „Umsichtig“, „Vorsätzlich“ und „Unbeabsichtigt“ eingeteilt wird. Dabei hebt dieser Artikel besonders hervor, dass *Technical Debt* nicht immer nur schlechte Einflüsse besitzt, sondern nur dann, wenn sie unbeachtet bleibt und die Schulden nicht zurückgezahlt werden.

In der Arbeit von Brown *et al.* werden die Eigenschaften der *Technical Debt* mit insgesamt sieben Punkten beschrieben: Visibility, Value, Present Value, Debt accretion, Environment, Origin of debt, Impact of debt. Insgesamt alles Eigenschaften, die genauer beschreiben, welche Einflüsse die *Technical Debt* auf das System hat und welchen Wert es hervorruft. Aber auch wenn diese Eigenschaften treffend das Phänomen der *Technical Debt* darstellen, bleibt die Antwort nach der Quantifizierung der Eigenschaften in dem Artikel offen und damit vorerst ungelöst. Dies erkennen die Autoren ebenfalls und widmen den Problemen bei dem Erfassen von Messwerten einen eigenen Abschnitt, der jedoch nur Erklärungen und keine Antworten oder Lösungsansätze liefert.

Ebenfalls erfasst dieser Artikel, dass weite Bereiche der *Technical Debt* noch offene Fragen aufwerfen. Die vier klassifizierten Felder sind laut dem Artikel folgende: Refactoring opportunities, Architectural issues, Identifying dominant sources of debt, Measurement issues. Die Autoren beschreiben bei allen Feldern den Einfluss sowie die Wichtigkeit, mit der das Forschungsfeld begründet wird.

Interessant dabei ist der letzte Punkt, die *Measurement issues*, welcher die Fragen aufwirft, wie man die *Technical Debt* messen können müsste und welche Einflüsse auf die Kosten für Eliminierung, Fehlerbehebung und Maintenance zustande kommen.

Zuletzt fasst der Artikel noch zwei Fragen auf, welche umso interessanter sind, sobald die oben beschriebenen Fragestellungen gelöst sind: a) Was wird sich verändern? und b) Wen wird es interessieren?

Der Artikel hat sein Ziel erreicht, die offenen Forschungsfelder der *Technical Debt* zu identifizieren und eventuelle Probleme zu nennen. Damit ist er ein guter Ansatz für diese Diplomarbeit gewesen, um ein Verständnis für dieses Phänomen zu erlangen und Parallelen zur Redundanz in Systemen zu ziehen. Jedoch geht der Artikel auch nicht weiter, als diese Fragen aufzustellen. Auch ist bislang nur wenig Neues in den angesprochenen Feldern erschienen, was an der Komplexität derselbigen, am Desinteresse der Forschung oder kommerzielle Gründe haben kann. Am Ende hört der Artikel genau dort auf, wo die Thematik dieser Diplomarbeit beginnt.

Graph annotations in modeling complex network topologies Ein Artikel mit etwas anderer Ausrichtung ist von den Autoren Dimitropoulos *et al.* [14]. Der Bezugspunkt dieses Artikels zu dem Thema dieser Arbeit besteht darin, dass hier versucht wird, ein Netzwerk anstelle mithilfe eines ungerichteten Graphen als bidirektionalen, annotierten Graphen darzustellen. Dies ist ein besonders wichtiger Punkt, da in dieser Diplomarbeit mit solcher Art von Graphen hantiert wird, um Systemeigenschaften formal darstellen zu können sowie daran Berechnungen durchführen zu können. Wie genau die Graphen für die Architektur von Softwaresystemen verwendet werden, wird in den folgenden Kapiteln dieser Arbeit genauer beschrieben.

Die Autoren beziehen sich hauptsächlich auf Annotationen eines Graphen, um autonome Systeme zu beschreiben. Dabei spiegeln diese Annotationen diverse Eigenschaften wider, welche Konnektivität oder Bandbreite des Systems darstellen. Sie argumentieren damit, dass eine bloße Reproduktion einer Struktur in einem Graphen nicht aussagekräftig ist und wichtige Informationen dadurch verloren gehen.

Für diesen Zweck formalisieren sie in dem Artikel die Annotationen an Kanten des Graphen und lassen die Annotationen an Ecken aus Gründen der Übersichtlichkeit heraus. Die Formalis-

men selbst sind sehr exakt mathematisch gehalten und beziehen sich unter anderem auf Werke von anderen Netzwerkanalysen [36] sowie Mechaniken von gefärbten Graphen [50].

Der Hauptteil des Artikels handelt das Thema der Konstruktion eines synthetischen Graphen ab. Nachdem die Autoren sich geeignete Metriken aus bereits existierenden Graphen extrahiert haben (unter anderem Degree distribution, Annotation distribution, usw.), nutzen sie diese Daten um damit einen neuen Graph als Netzwerk-Simulation zu bauen. Ihre dazu benötigten drei Schritte sind Extraction, Rescaling und Construction. Ihre anschließenden Messungen mit dem neuen Modell zeigen auf, dass es sich durchaus lohnt, Annotationen und daraus resultierende Informationen bei der Generierung von Graphen einzubeziehen.

In Bezug auf diese Arbeit wurde von diesem Artikel, bedingt durch den grundsätzlich verschiedenen Anwendungszwecken, nur recht wenig Einfluss ausgeübt, alleine wegen der unterschiedlichen Gestaltung des strukturierten Graphen. Jedoch gibt er Aufschluss über die Wahl eventueller Graph-Metriken, Formalismen und Berechnungen an Graphen mit Annotationen.

Zukunftsfähige Softwaresysteme Der Artikel von Furrer [20] beschreibt auf neun Seiten den Kontext dieser Arbeit genauer: Softwaresysteme und deren Notwendigkeit, sich neuen Anforderungen effektiv und effizient anzupassen. Der dafür vorgestellte Mechanismus ist die *Managed Evolution*, welche nach einer kurzen Einführung über zukunftsfähige Software genauer erläutert wird. Allgegenwärtig wird das Thema der *Technical Debt* und *Architecture Erosion* angeschnitten.

Inhaltlich ist der Artikel stark mit [19] und [41] verwandt, was auf die gemeinsame Autorschaft zurückzuführen ist. Jedoch kann man den Artikel all jenen empfehlen, welche keine Gelegenheit für Vorlesungsskripte oder ein ganzes Buch über die Thematik finden. Da die beiden anderen Quellen jedoch ausführlicher die Thematik der Evolution und Architektur eines Softwaresystems behandeln, wird in dieser Arbeit vorwiegend auf die beiden anderen verwandten Quellen Bezug genommen. Viele Grafiken und Thesen, die in allen drei Werken vorkommen, sind auch in dieser Arbeit referenziert vorzufinden.

An Empirical Model of Technical Debt and Interest Der Artikel von Nugroho *et al.* [42] ist deswegen interessant, da die Autoren versuchen, ein Modell zu entwickeln, welches die *Technical Debt* in Zusammenhang mit den entstehenden Kosten (Fixkosten und Maintenance-Kosten) bringt. Dass dafür ein empirisches Modell benutzt wird, was in der Intention dieser vorliegenden Arbeit eher ausgeschlossen ist, macht diesen Artikel jedoch nicht uninteressanter. Vielmehr wird dadurch eine vermutlich bessere Methode erreicht, die tatsächlichen Kosten der *Technical Debt* im System einzuschätzen, jedoch liefert die empirische Grundlage der Methode keine direkte Kausalität.

Der Artikel ist in drei Abschnitte eingeteilt, wovon der erste Abschnitt genauestens erläutert, inwiefern sich die *Technical Debt* definieren lässt. Dabei Unterscheiden die Autoren die Kosten für die *Technical Debt* von den normalen Maintenance-Kosten, welche in einem System anfallen. Der Bezug besteht dabei immer zum Ideal-System. Obwohl die Klassifizierung zwischen gewollter und ungewollter *Technical Debt* angesprochen wird, wird darauf in weiteren Verlauf nicht eingegangen. Die Schuld wird nur durch die „Lücke“ zwischen dem aktuellen und dem Ideal-System definiert. Im zweiten Block werden dir durch *Technical Debt* entstandenen Kosten durch Metriken auf Code-Ebene ermittelt, welche das System in eine von fünf verschiedenen Kategorien einteilt. Basierend auf der derzeitigen Kategorie und der höchsten Kategorie wird der Unterschied in Entwickler-Monaten aufgezeigt, den es benötigt, um das System dem Ideal-System anzunähern. Grundsätzlich basiert das genutzte Modell dazu auf der Anzahl aller Code-Zeilen

(*LOC*), es wird aber ausdrücklich darauf hingewiesen, dass auch funktionale Größen (wie *Function Points*) genutzt werden können. Sobald das System in eine der Kategorien eingeteilt wurde, wird mit mehreren Faktoren die tatsächliche Kostenrechnung angestellt. Dabei wird das System dahingehend analysiert, wie viel Prozent des Codes wie stark geändert werden müssen, und was dies unter den derzeitigen Umständen kostet. Die ermittelten Kosten für die *Technical Debt* geben einen jährlichen Betrag an (inklusive einer jährlichen Wachstumsrate), inwiefern man mehr für die Maintenance mit und ohne *Technical Debt* ausgeben muss. Der letzte Block im Artikel beschreibt noch einmal genauer das Vorgehen bei einem Beispielsystem.

Obwohl in dem Artikel eine andere Intention anliegt, und zwar die reine (empirische) Quantifizierung von *Technical Debt* im System, ist er doch sehr verwandt mit der vorliegenden Arbeit. Oft werden die gleichen Schlüsse gezogen und beachtet, wie zum Beispiel Parameter für die Erfahrung der Entwickler und den Zugriff auf Technik, dem Bezug zum Ideal-System oder die inhaltliche Größe von Systemen. Die Unterscheidung zwischen Maintenance-Kosten und Kosten für *Technical Debt* ist zwar möglich, allerdings muss hinzugefügt werden, dass *Technical Debt* immer ein Teil der Maintenance bleiben wird, weswegen in dieser Arbeit auch die *Technical Debt* später in die drei Kategorien *Adaptive*, *Corrective* und *Preventive Maintenance* aufgeteilt wird. Damit lassen sich Ursachen und Parameter besser kausal erläutern. Zudem wird in dem Artikel kein statisches Modell benutzt sondern auf Basis von jährlichen Kosten abgerechnet. Auch hier gilt wieder: dies ist gut für eine genaue empirische Ermittlung der Kosten, jedoch ungeeignet für die Darstellung in einem kausalen Zusammenhang. Jedoch ist dieser Artikel dahingehend wichtig, da dieser am ehesten den „praktischen“ Anteil von dieser Arbeit darlegt. Eine gute Kombination aus den Ideen beider Werke ist damit durchaus vorstellbar.

Weitere Arbeiten Weiterhin gibt es noch einige Artikel und Referenzen, welche ebenfalls oft die Thematik mit dieser Arbeit tangieren, und ebenfalls noch erwähnt werden sollten.

In Curtis *et al.* [12] wird ebenfalls ein empirisches Modell für Berechnungen der *Technical Debt* aufgezeigt. Des Öfteren wird hierbei mit empirisch ermittelten Werten aus [5] gehandhabt, was zwar für eine Kosteneinschätzung möglich ist, aber für einen kausalen Zusammenhang leider nicht hinreichend ist. Des Weiteren wird auch in Letouzey [32] eine Methode vorgestellt, die *Technical Debt* in einem System zu erfassen und einzuschätzen. Dabei stellt das *SQALE*-Modell eine Einordnung und Einschätzung des Systems in verschiedene Qualitätseigenschaften dar, welche dann pyramidenartig analysiert werden. Der Artikel Zegura *et al.* [68] betrachtet ausführlich verschiedene Methoden, die Topologie des Internets in einen Graph zu fassen, zusammen mit den jeweiligen Problemen und Berechnungen des Graphen. Relevant ist auch Oquendo *et al.* [43], wo mehrere Themen zu Softwarearchitektur als Report zusammengefasst sind, unter anderem zu Architektur-Patterns, UML, Entwicklungen, Modellen und mehr. Weiterhin ist die Arbeit von Carzaniga *et al.* [7] noch dahingehend interessant, wie das Problem von Code-Redundanz angegangen wird. Hierbei wird ein Äquivalenz-Modell auf dem Code konstruiert und abstrahiert, sodass ein Vergleich und Messen nach Redundanz möglich ist, selbst wenn der Code nicht exakt gleich ist oder sich der Output minimal ändert. In Ahn *et al.* [1] wird ein Modell vorgestellt, welches den Anspruch erhebt, die Kosten für Software-Maintenance auf Basis von *Function Points* einzuschätzen. Das Modell setzt auf die Methodik, die „normal“ errechneten *Function Points* zu adjustieren und daran den geschätzten Aufwand für Maintenance festzustellen. Zum Schluss sei noch Chapin *et al.* [8] zu nennen, da hier die Begrifflichkeiten zu Software-Evolution und -Maintenance noch einmal genauer, wenn auch aus einer anderen Perspektive, angegangen werden.

1.5 Begriffe und Erläuterungen

Innerhalb dieser Arbeit wird es mehrere Begrifflichkeiten geben, welche benötigt werden, um den tieferen Zusammenhang nachverfolgen zu können. Es werden hier kurz die wichtigsten Elemente erläutert, welche später als gegeben vorausgesetzt werden können. Zudem wird es unumgänglich sein, diverse englischsprachige Begriffe zu verwenden, da diese entweder zu stark in der IT-Welt etabliert sind oder sich kein geeignetes Wort in deutscher Übersetzung finden lässt. Wenn nötig, werden in dieser Arbeit benutzte englische Fachwörter bei dem ersten Auftauchen kurz erläutert. Es wird ebenfalls darauf geachtet, dass eventuelle Formalismen nur dann benutzt werden, wenn diese später nötig sein werden, zum Beispiel bei Definitionen und speziellen Aussagen.

Komponente oder auch Software-Komponente c, stellt in dieser Arbeit ein Modul, eine Klasse, ein Strukt oder ähnliche Kapselungen von Funktionen oder Methoden dar. Dabei spielt die eigentliche Größe keine Rolle. Wichtig ist hier zu definieren, dass eine Komponente immer nur über Interfaces agiert und ihre inneren Abläufe nach außen hin versteckt. Die Interfaces selbst sind dabei parametrierbar, sodass angenommen werden kann, dass eine Komponente auf den für ihren Zweck gedachten Weg angesprochen werden kann.

Die Komponenten haben ihrerseits Beziehungen zu anderen Komponenten, welche sie für ihren eigenen Ablauf benötigen. Ebenso besitzen sie Beziehungen zu Datenstrukturen, welche die (externen) Daten bereithalten, welche die Komponente ebenfalls für ihren korrekten Ablauf benötigt.

Die Menge aller Komponenten in einem System wird hier mit $\mathbb{C}$ bezeichnet.

Datenbestand oder Datenbank d ist in dieser Arbeit einfach definiert als Datensammelpunkt. Dazu zählen alle Datenbanksysteme und Dateien, welche externe Daten für Komponenten in irgendeiner Weise bereit halten. Ein Datenbestand gilt immer genau dann als ein alleinstehender Datenbestand, wenn er keine inhaltlichen Beziehungen zu anderen Datenbeständen besitzt. Dabei spielt es keine Rolle, ob ein Datenbestand aus einem kompletten DBMS besteht, ein von anderen Daten separierter Tablespace ist oder nur aus kopierten und hart-codierten Werten im Quellcode eingearbeitet ist. Sobald die Daten eines Datenbestandes nichts mehr mit anderen Daten zu tun haben, gilt dieser als ein eigener, angeschlossener Datenbestand. Der Übersichtlichkeit halber wird dann in dieser Arbeit von einer eigenen Datenbank pro Datenbestand ausgegangen, da es für das Thema und das Ziel dieser Arbeit nicht relevant ist, ob sich zwei Datenbestände in einer oder mehreren Datenbanken befinden.

Ein Datenbestand besitzt ebenfalls Beziehungen zu anderen Datenbeständen, wenn sich deren Daten in einer redundanten Weise überschneiden. Wie genau dies modelliert werden kann, wird in späteren Kapiteln erläutert. Jedoch wird der Datenbestand niemals eine ausgehende Beziehung zu einer Komponente anbieten, da die Komponenten diejenigen sind, welche den Datenbestand durch Interfaces nutzen und nicht umgekehrt.

Die Menge aller Datenbestände in einem System wird hier mit $\mathbb{D}$ bezeichnet.

Beziehung ist eine Verknüpfung zweier Knoten, in diesem Fall eine Verbindung zwischen und innerhalb der Mengen $\mathbb{C}$ und $\mathbb{D}$. Eine Beziehungen wird mit $x_i \in \mathbb{X}$ notiert, jedoch nur sehr selten benutzt, da man hauptsächlich Komponenten und Datenbestände nutzen wird.

Element ist ein Teil des Systems mit $e \in \{\mathbb{C} \cap \mathbb{D} \cap \mathbb{X}\}$ und damit entweder eine Komponente c, ein Datenbestand d oder eine Beziehung x.

System ist der Oberbegriff für die Gesamtmenge $\mathbb{E}$ aller Elemente e sowie deren Beziehungen zueinander. Dazu zählen alle bereits vorhandenen Elemente, aber auch durch eine Erweiterung hinzugefügten Elemente, sowie all deren Beziehungen. Insgesamt ist davon der (später definierte) Graph des Systems abzugrenzen, welcher mit G notiert wird.

Erweiterung eines Systems bedeutet jegliche Änderung am bestehenden System. Dabei besitzt die Erweiterung immer mindestens ein Element, das heißt eine Komponente, einen Datenbestand oder eine neue Kante kann aber jeweils beliebig viele aufweisen. Dabei gilt es, alle in der Erweiterung neu hinzugefügten Elemente mit dem bestehenden System wie vorgesehen zu verbinden. Wie sich das genau im Modell äußert, darauf wird ebenfalls in den späteren Kapiteln genauer eingegangen.

Als Notation eignet sich hier $e^+ \in \mathbb{E}_x^+$ mit $\mathbb{E}_x^+ = \{\mathbb{C}_x^+ \cap \mathbb{D}_x^+ \cap \mathbb{X}_x^+\}$ und damit $e^+ = c^+ \vee d^+ \vee x^+$ als Beschreibung für ein Element e^+ der bestimmten Erweiterung $\mathbb{E}_x^+$ des Systems mit einer Menge an geänderten Komponenten $\mathbb{C}_x^+$, geänderten Datenbeständen $\mathbb{D}_x^+$ sowie deren Beziehungen $\mathbb{X}_x^+$ in der Erweiterung mit dem Index $x \in \mathbb{N}$. Und mit dieser Definition ergibt sich daraus auch $\mathbb{E} = \{\mathbb{E}_0^+ \cap \mathbb{E}_1^+ \cap \ldots \cap \mathbb{E}_n^+\}$ mit $n \in \mathbb{N}$ als Index der jüngsten Erweiterung.

DevC und TtM bezeichnet die beiden Hauptwerte *Development Cost* und *Time to Market*, die beiden Hauptattribute bei der Einschätzung des Wertes eines Systems. Während *DevC* die Entwicklungskosten für Manager, Entwickler, Technik und weiteres beinhaltet, bezeichnet *TtM* die Zeit von Beginn der Planung bis zur Marktreife des Produkts oder des Features. Beide Werte sollten gering gehalten werden, da sie ausschlaggebend für die Investition und den Aufwand in einem Softwaresystem sind und in betriebswirtschaftlichen Bereichen eine Rolle spielen.

Function Points ist eine abstrakte Einheit, die den inhaltlichen Umfang einer Komponente oder eines Systems widerspiegelt. Es gibt mehrere Verfahren, welche aus den Anforderungen und den Funktionen eines Systems die *Function Points* ermitteln [16] [38]. Insbesondere sind die *Function Points* für eine sinnvolle Schätzung der Kosten innerhalb der Projektplanung relevant. In dieser Arbeit werden die *Function Points* als eines der grundlegenden Elemente angesehen, um die angesprochene Quantität im Nachweis herzustellen, wie noch in späteren Kapiteln aufgezeigt wird. Dabei wird sich der Umstand zunutze gemacht, dass *Function Points* relativ unabhängig der genutzten Technologien ermittelbar sind und zusätzlich eine Einschätzung für *TtM* und *DevC* liefern, da bereits einige Statistiken vorliegen, wie viel Zeit und Geld pro *Function Point* für ein System aufgebracht werden muss, was wiederum dem Zweck dieser Arbeit dienlich ist.

2 Grundlagen von Softwaresystemen

2.1 Eigenschaften

Softwaresysteme sind heutzutage in fast jedem Bereich der Wirtschaft vertreten. Dabei reichen deren Einsatzgebiete von einfachen Kassensystemen für simple Abrechnungen bis hin zu komplexen Maschinerien, auf welchen das gesamte Business aufgebaut ist. Um diese Systeme besser dafür zu katalogisieren, inwieweit dessen Architektur Wert-bestimmend ist, sollen die folgenden Kapitel die Grundsteine und Hauptbegriffe dahingehend näher erläutern. Dabei wird auf verschiedene Aspekte eingegangen, die in dem späteren Verlauf und dem dazugehörigen Nachweis des Wertes in dieser Arbeit von Nutzen sind.

2.1.1 Ausmaße und Dimensionen

Einzelne Softwaresysteme sind in verschiedenster Weise in die Unternehmen integriert und werden auch auf komplett unterschiedliche Weisen gehandhabt. Dabei unterscheidet man von den einfachen IT-Systemen (*standard*), den großen IT-Systemen (*large*) und den sehr großen IT-Systemen (*very large*). Was genau ein sehr großes System von den anderen unterscheidet, dem ist in Murer *et al.* [41] ein eigenes Kapitel gewidmet. Das System selbst hat damit auch verschiedenste Auswirkungen und genießt unterschiedliche Prioritäten, was dessen Evolution sowie Management betrifft. Während bei einem einfachen System, welches durchaus gar keine Eigenproduktion des eigenen Unternehmens sein muss und zugekauft worden sein kann, die Interessen der Weiterentwicklung meist eher nur gering sind und aus neuen notwendigen Anforderungen erfolgen, sind im Gegensatz dazu die Interessen bei Großprojekten meist der Natur des wirtschaftlichen Vorsprungs und damit des eigenen Antriebs des Unternehmens, neue Funktionen zu entwickeln und das System voranzubringen. Dabei kann der eigene Vorsprung gegenüber Konkurrenten eine bedeutende Rolle spielen.Solch eine Motivation zur Weiterentwicklung des Systems hätte ein kleines Unternehmen nicht, welches das System nur einsetzt, um vordefinierte Tasks zu automatisieren.

Ein großes IT-System weiterzuentwickeln erfordert ein hohes Maß an Analyse, Management und Koordination. Ungehemmte Weiterentwicklung wird mit sehr hoher Wahrscheinlichkeit in einem „Fossil-System“ enden [19] [20], welches entweder gar nicht oder nur mit extrem hohen Aufwand weiterentwickelt werden kann. Auf solche Art und Weise entstehen dann die sogenannten *Legacy Systems*, also Systeme, welche ihre Entstehungszeit weit hinter sich haben und seit vielen Jahren und Jahrzehnten nur weiterentwickelt und ausgebaut wurden. Das bedeutet allerdings nicht, dass diese *Legacy Systems* ungeeignet sind. Jedoch erfordert es ein hohes Maß an Management, dass solche Systeme weiterhin benutzbar bleiben. Dabei ist das Management der System-Evolution logischerweise bei großen IT-Systemen erheblich wichtiger als bei kleinen Systemen.

Ein Softwaresystem kann verschiedenste Eigenschaften aufweisen, die definieren, welches Ausmaß oder Dimension es annimmt. Das dabei bekannteste Maß ist die simple Anzahl an Codezeilen ($\#LOC$) für das gesamte System. Während bei einem einfachen Buchungssystem in einem

Handwerksbetrieb oder einer Handy-App gerade mal einige Tausend davon das System definieren, nimmt dies bei Großfirmen wie Autoherstellern, Banken oder großen Service-Providern mit vielen Millionen Zeilen pro System komplett andere Dimensionen an, sehr anschaulich in [39] dargestellt. Beispielsweise seien hieraus eine durchschnittliche Handy-App mit 200'000 *#LOC*, der *Large Hadron Collider (LHC)* in Bern mit knapp 50 Millionen *#LOC* und die durchschnittliche Auto-Software mit 100 Millionen *#LOC* genannt (siehe dazu auch [9]). Häufig zählen auch die Anzahl an Vorgängen und Tasks im Unternehmen, welche ohne das System nicht durchführbar wären. Während bei einfachen Systemen nur „normale" Vorgänge automatisiert werden und im Notfall auch per Hand erledigt werden können (beispielsweise eine Abrechnung einer Auto-Reparatur), ist die Abhängigkeit bei sehr großen Systemen meist fundamental, da hierbei fast jeder Prozess von der Verfügbarkeit des Systems abhängt (beispielsweise das Buchen von Flügen einer Airline). Damit ist auch die Abhängigkeit des Unternehmens von dem jeweiligen System begründet sowie die daraus resultierende Motivation, das System einer sinnvollen Evolution zu unterziehen und das System zukunftsfähig zu halten.

Die Art von Systemen, welche die meiste Evolution und das größte Management benötigen, und damit auch die Motivation dieser Arbeit treffen, sind die sehr großen Systeme. Es gibt einige grundlegende Punkte, an denen sich ein großes oder sehr großes System von normalen unterscheidet. In Murer *et al.* [41] ist ein sehr großes Softwaresystem definiert mit folgenden neun Eigenschaften:

1. *Complexity* als die Eigenschaft, welche die überdurchschnittliche Komplexität in solchen Systemen beschreibt. Dieser Wert ist bei sehr großen Systemen überdurchschnittlich hoch.

2. *Functional Size* als ein Wert, der beschreibt, wie viel Funktionen oder Arbeitsaufwand in dem System enthalten sind. Die Autoren betiteln ein System als *very large*, wenn es mehrere Millionen *LOC* beinhaltet.

3. *Legacy Code* ist der Anteil von der *Functional Size*, welcher noch aus den Entwurfszeiten unmodifiziert enthalten ist. Auch hier ist eine generelle Orientierung gegeben: ein sehr großes System wird ohne Restrukturierung nicht länger als zehn Jahre effizient arbeiten können.

4. *Rate of Change* als die Eigenschaft, welche die Häufigkeit an Änderungen am System beschreibt. Diese hat direkt Auswirkungen auf die Komplexität und Eigenschaften des ganzen Systems und darin enthaltener Komponenten.

5. *System Value* ist der wohl wichtigste ökonomische Wert. Während langer Laufzeit werden viele Änderungen am System vorgenommen und neue Funktionen implementiert, um den Wert des Systems mindestens zu erhalten. Bei sehr großen Systemen ist hier dieser Wert direkt zusammenhängend mit den Kosten für Änderung und Ersatz, die durchaus in die Milliarden gehen können.

6. *Mission Criticality* ist der Wert, inwieweit das Unternehmen noch agieren kann, sollte das System unerreichbar sein. Bei Unternehmen, welche sehr große Systeme benutzen, ist dies meist direkt eine Frage nach der Verfügbarkeit, da, sobald die Verfügbarkeit eingeschränkt ist, das Unternehmen nicht mehr handlungsfähig ist. Damit ist das System und dessen Stabilität und Verfügbarkeit ein kritischer wirtschaftlicher Faktor.

7. *Diversity of Technology Base* ist eine Angabe zu den unterschiedlich benutzten Technologien in einem System. Es liegt in der Natur von sehr großen Systemen, dass hier viele

unterschiedliche Technologien angesprochen werden, sowohl im Frontend als auch im Backend.

8. *Governance* als ein Wert für Zuständigkeiten ist in einem sehr großen System ebenfalls komplexer als normal. Dabei können einzelne Komponenten unterschiedliche Verwaltungen haben und umgekehrt der Zugriff auf einige Komponenten bestimmten Personen verwehrt bleiben. Eine zentrale Organisation des Systems ist meistens nicht mehr gegeben.

9. *Organization* ist ähnlich wie *Governance* ein Wert der externen Struktur. Aber anders als bei *Governance* wird hier nicht die Zuständigkeit oder Kontrolle angesprochen, sondern vielmehr der Besitz einzelner Komponenten. Denn auch dieser Wert ist in einem sehr großen System höher als normal und verursacht unter Umständen Probleme bei der Evolution. Schließlich müssen verschiedene Besitzer verschiedener Komponenten den Änderungen auch zustimmen.

Abschließend sei hier erwähnt, dass es nicht ausschlaggebend ist, welches Unternehmen ein System benutzt oder in welcher Branche es tätig ist, sondern vielmehr auf welche Art und Weise das Unternehmen in das „Investitionsgut“ Software investiert und davon abhängig ist. Dabei ist es wichtig zu erkennen, dass große Systeme mehr *Maintenance* („Pflege“) benötigen als kleinere und dahingehend auch anfälliger für schlechte Umsetzungen sind, sodass der Verfall von Struktur und Wert schneller und kritischer verläuft. Dies gilt besonders für von dem System abhängige Unternehmen.

2.1.2 Die Kosten eines Systems

Ein wichtiger Punkt bei der Evaluation von Softwaresystemen ist der Kostenfaktor. Dieser ist bestimmend, ob es sich lohnt, ein System zu bauen, ein System zu erweitern oder ein System zu ersetzen. Dabei setzten sich diese Kosten aus den unterschiedlichsten Kategorien zusammen, welche sich in allen Phasen des Lebenszyklus eines Softwaresystems wiederfinden.

Die Kosten, um welche es sich in dieser Arbeit handelt, sind nicht mehr die, welche in der Entwicklung des Systems entstanden sind. Vielmehr wird vorausgesetzt, dass die Entwicklungsphase des Systems bereits in der Vergangenheit liegt, und ein fertiges System vorliegt sowie dessen Entwicklungskosten bereits getilgt sind. Jedoch ist damit erst der Grundstein der Evolution eines Systems gelegt, und damit fallen auch in Zukunft weitere Kosten an. Dabei kann man die Bereiche, in denen ein existierendes Softwaresystem Kosten verursacht, in folgende einteilen [24] [33]:

1. *Adaptive Maintenance (Evolution)* als der Teil, der Kosten verursacht, wenn komplett neue Anforderungen und Funktionen das bestehende System erweitern sollen. Diese beinhalten die Arbeit der Entwickler, der Erwerb sowie das Integrieren neuer Komponenten und das Entwerfen und Durchführen von Tests mit abschließendem Qualitätsmanagement.

2. *Corrective Maintenance* ist der klassische Fall der Fehlerbehebung im System. Das beinhaltet größtenteils die Kosten für die Entwickler, die den Fehler suchen und entfernen.

3. *Preventive Maintenance* (oder auch *Perfective Maintenance*) ist der Bereich, in dem alte Teile des Systems gegen neue Ausgetauscht werden sollen. Entweder, weil dessen Funktionalität mangelhaft geworden ist oder neue Technik zum Einsatz kommen soll. Diese Kategorie kann man nochmals unterteilen in die folgenden beiden Unterbereiche:

a) *Refactoring* ist die Ersetzung von strukturellen Teilen des Systems, zum Beispiel der Austausch einer Komponente oder das Verwenden eines anderen Protokolls.

b) *Rearchitecting* ist gröber gefasst und bedeutet das Ersetzen von ganzen Teilen des Systems im Sinne der Architektur. Das beinhaltet zum Beispiel die Umstellung von einer Client-Anwendung auf eine Web-Applikation.

Die *Preventive Maintenance* dient ebenfalls der Beseitigung von *Architecture Erosion* und der *Technical Debt* (siehe dazu mehr in Kapitel 2.3 „Fehler durch Architektur und Management“). Dabei werden häufig alte Komponenten, welche nicht mehr sinnvoll sind zu unterhalten, gegen neue ersetzt, welche die gleiche Funktionalität bieten, um zukünftige Evolution zu vereinfachen.

4. *Operativ* ist kein „echter“ Kostenbereich, tritt aber als Kategorie mitunter auf. Dies ist der Teil an Kosten, welcher anfällt, wenn während des Betriebs des Systems Fehler auftreten oder Mehraufwand betrieben werden muss. Das beinhaltet nicht direkt die Kosten der Fehlerbehebung (was in die *Corrective Maintenance* zählen würde) sondern die Folgen des Fehlers im operationalen Einsatz, zum Beispiel unnötig längere und kompliziertere Prozesse durch Administration des Systems, Strafzahlungen wegen Nicht-Einhaltung von Qualitätsattributen oder Rückerstattungen von Zahlungen an Kunden, aber auch Reputationsverlust des Unternehmens, was nur noch schwerlich formal messbar ist. Dieser Kostenbereich ist hier nur der Vollständigkeit halber erwähnt, wird ab hier aber nicht weiter unter den normalen (Aufwands-)Kosten des Systems gezählt.

Um die Kosten eines Systems in diesen einzelnen Bereichen zu beziffern, müssen verschiedene Faktoren einbezogen werden und grundlegende Fragestellungen vorher beantwortet werden. Dabei ist der Begriff *Kosten* weiter zu fassen als nur der reine Betrag in einer realen Währung. Im Bezug auf Softwaresysteme sind Kosten in *Entwicklungskosten DevC* [54] sowie in *Entwicklungszeit TtM* [59] zu benennen. Dabei hängen beide Faktoren zusammen und bedingen sich gegenseitig. Beide treten in allen vier oben genannten Bereichen der Kosten auf. Während alle Bereiche ihre Kosten rechtfertigen, kann ein Unternehmen jedoch diese extrem erhöhen, indem schlechtes Management betrieben wird. Es ist jedoch gängige Meinung, dass eine gute Evolution des Systems in allen vier Bereichen die Kosten verringert, da dies beide Kostenfaktoren für die neue Funktionalität selbst, dessen Integration und operationales Verhalten direkt auf ein Mindestmaß beschränkt.

Der letzte Fall, der nicht in den Kostenbereichen auftritt, ist gleichzeitig der Extremfall. Die Situation, dass die Architektur und das Management des Softwaresystems solange außer Acht gelassen wurde, dass das System zu einem *Fossil System* gewandelt ist. Dabei wurde jegliche Evolution nur dadurch etabliert, neue Funktionen einfach und bestmöglich optimiert für beide Kostenfaktoren in das bestehende System zu integrieren. Das Ende des Ganzen ist ein System, welches eine extrem hohe Komplexität besitzt und gleichzeitig so hohe Kosten in allen drei Bereichen aufweist, dass die Evolution des Systems in eine Sackgasse geraten ist [41]. Wenn dann das betroffene Unternehmen von dem System abhängig ist, dann droht im schlimmsten Fall eine Insolvenz des Gewerbes. Da dieser Fall nicht mehr in Kostenfaktoren messbar wäre und einfach einen kompletten Wertverlust des Unternehmens beinhaltet, ist dieser nicht in den normalen Kostenbereichen aufgeführt. Aber auch eine teilweise Insolvenz oder ein allgemeiner Wertverlust eines Unternehmens ist nicht in näherer Betrachtungsweise und daher auch nicht im Umfang dieser Arbeit enthalten. Jedoch sollte man immer bedenken, dass ein Verfall eines Systems durchaus auch auf das gesamte Unternehmen Auswirkungen haben kann.

2.1.3 Der Wert eines Systems

Gleichzeitig zu den anfallenden Kosten im Unternehmen bei der Unterhaltung eines Softwaresystems kann solch ein System den eigenen Wert (= Geschäftsnutzen, engl. *Business Value*) und damit unmittelbar den Wert des Unternehmens immens erhöhen. Hier begreift man das System wieder als „Investitionsgut“ einer Firma, welches es zu managen gilt. Durch fortschrittliche Entwicklung und Evolution dieser Systeme kann ein Vorteil auf dem Markt gewonnen werden. Jedoch kann man den Wert nicht direkt in Kategorien wie bei Kosten einteilen. Der Wert des System ergibt sich eher indirekt, durch Kosteneinsparung, durch Prozessverkürzungen und erweiterten Handlungsspielraum, was sich dann wiederum auf unternehmensspezifische Möglichkeiten auswirkt und im Endeffekt dessen Wert steigern kann.

Dabei ist der wichtigste Punkt, ob der Wert des Systems auch die Kosten des selbigen decken kann. Mit guter Architektur, gutem Design, guter Programmierung sowie geregelter Evolution des Systems ist der Wert meist gegeben. Umgekehrt bewirkt eine Vernachlässigung einer dieser Punkte einen Verfall der Software, welcher auf längere Sicht die Kosten über das Niveau des Wertes ansteigen lässt. Gerade diese Wechselwirkung wird sich zunutze gemacht, über anfallende Kosten, beziehungsweise deren Vermeidung, den Wert des Systems festzustellen. Diese invertierte Berechnung wird ebenfalls der Ansatz in dieser Arbeit werden, um kausal den Wert eines Systems bezüglich der Evolutionsfähigkeit nachzuweisen.

Eine weitere wichtige Eigenschaft für den Wert einer Software ist neben dessen Geschäftsnutzens (dem *Business Value*) auch dessen Zukunftsfähigkeit [41]. Ein System, welches nicht weiterentwickelt wird oder werden kann, ist nach einigen Jahren nicht mehr sachgemäß nutzbar. Dies wird durch sich ändernde Technologien (Plattformen, Programmiersprachen), Ideale (Programm-Design), Anforderungen und Funktionen oder auch Regeln und Richtlinien bedingt, welche das Unternehmen umsetzen muss. Eine geregelte Evolution kann dem aber entgegenwirken, womit der Wert des Systems beibehalten oder gar gesteigert wird. Die beiden Punkte dafür, welche maßgeblich den Werteanstieg bezüglich der Zukunftsfähigkeit bestimmen, sind *Agility* (die Eigenschaft, sich neuen Anforderungen mit geringen *DevC* und *TtM* anzupassen) sowie *Resilience* (die Eigenschaft, resistent gegenüber Anfälligkeit aller Art zu sein). In Furrer [19] wird dazu genannt, dass die *Agility* für *„success in today's competitive markets“* und die *Resilience* für *„survival in today's dangerous environment“* benötigt wird. Diese beiden Werte kann man auch mit Metriken erfassen, jedoch ist es schwer, diese in einem System mehr als nur empirisch im Vornherein nachzuweisen. Näheres zu diesen Eigenschaften und deren Beziehungen findet sich in Kapitel 2.4 „Zukunftsfähige Softwaresysteme“. Am Ende der gesamten Kette aller Werte der geregelten Evolution bezüglich der Zukunftsfähigkeit steht das Unternehmen, welches durch einfache und unkomplizierte Erweiterung ihres Softwaresystems neue Funktionen und Anforderungen umsetzen kann, um damit auf dem umkämpften Markt standzuhalten, schnell zu reagieren und um davon anschließend zu profitieren.

Anhand von Abbildung 2.1 kann man die beiden verschiedenen Ausprägungen vom Wert der Software gut nachvollziehen. Auf der linken Seite befindet sich der klassische *Business Value*, welcher für die Zustimmung und Weiterentwicklung für das Projekt entscheidend ist. Anhand des *NPV (Net Present Value)* kann festgestellt werden, inwieweit sich das Projekt oder dessen Erweiterung lohnt und wirtschaftlich rentabel ist. Dieser Wert ist damit ein wichtiger Faktor für das Management des Unternehmens, welches sich entweder dafür oder dagegen entscheiden muss. Der Geschäftsnutzen ist dabei zweifach bedingt. Erstens muss der *NPV* stetig durch Investitionen in das System gehalten oder gesteigert werden, denn von alleine ergibt sich kein höherer Geschäftsnutzen. Durch gutes Management, Architektur und Entwicklung, Vereinfachungs- und Rearchitecting-Programme kann der Wert mindestens auf dem gleichen Level gehalten werden.

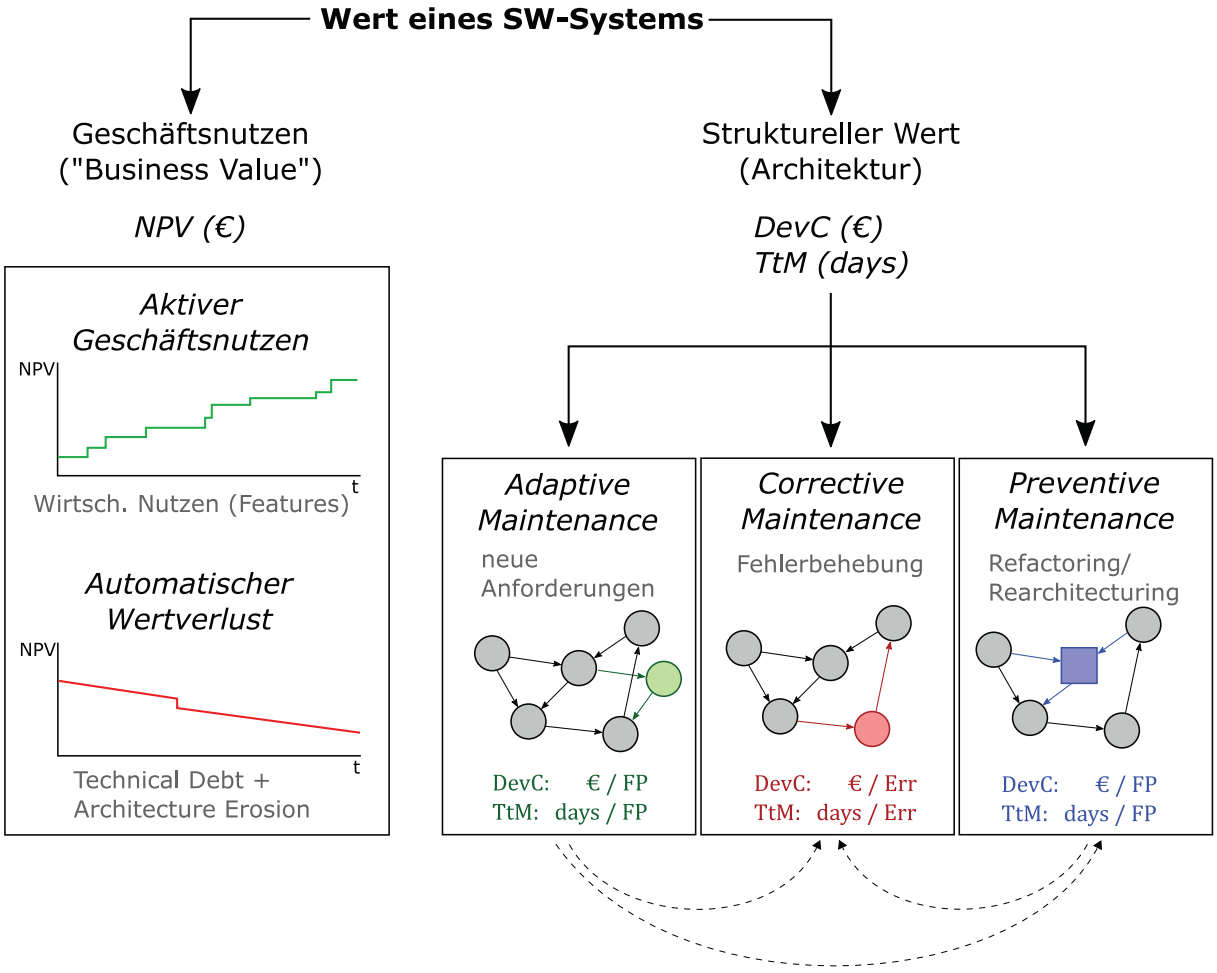


Abbildung 2.1: Der Wert einer Software.

Zweitens verliert das System automatisch an Geschäftsnutzen durch *Architecture Erosion* und *Technical Debt*. Steuert man nicht aktiv dagegen an, verliert damit das Gesamtsystem im Laufe der Zeit von selbst an Wert. Immer wenn Funktionen obsolet werden, Technologien veralten oder ein Konkurrent etwas Besseres anbietet, dann verliert das eigene System an Geschäftsnutzen durch *Architecture Erosion*. Sobald der eigene Managementprozess oder die Fahrlässigkeit von beteiligten Stakeholdern für Fehler im System sorgen, dann verliert das System durch *Technical Debt* ebenfalls an Geschäftsnutzen.

Der Wert der Software wird aber neben dem Geschäftsnutzen auch durch dessen Struktur generiert, wie in Abbildung 2.1 auf der rechten Seite zu sehen. Der strukturelle Wert kann durch die beiden Messwerte *DevC* und *TtM* festgelegt werden, da praktisch alle strukturellen Veränderungen mit diesen beiden Faktoren messbar sind. Anders als beim Geschäftsnutzen gibt es bei dem strukturellen Wert drei verschiedene Kategorien, welche alle bereits im Kapitel 2.1.2 „Die Kosten eines Systems“ genannt wurden: *Adaptive Maintenance*, *Corrective Maintenance* und die *Preventive Maintenance*. Bei der *Adaptive Maintenance* werden neue Anforderungen umgesetzt, in der Grafik durch die grüne Kugel markiert. Diese neue Anforderung resultiert in einem neuen strukturellen Element in der Software, welche eigene *DevC* und *TtM* generiert. Dabei wird für eine vergleichbare Größe auf eine Einheit gesetzt, welche den inhaltlichen Umfang

widerspiegelt, in diesem Fall *FP (Function Points)*. Möglich wären aber auch andere Einheiten wie *Object Points*, *Use-Case-Points* und Ähnliches. Daraus ergibt sich die Metrik an investierten Euros (oder Dollar) pro umgesetzten *FP* und investierte Tage pro *FP*. Ein System hat einen besonders hohen strukturellen Wert, wenn neue Erweiterungen günstig und schnell in das System eingebaut werden können. Eine gute Methode, die *Adaptive Maintenance* attraktiv zu halten und die Evolution des System in gute Bahnen zu lenken, ist die *Managed Evolution*, was später im Kapitel 2.4.3 „Managed Evolution“ weiter ausgeführt wird. Die zweite Kategorie ist die *Corrective Maintenance*, welche dann eine Rolle spielt, wenn durch Änderungen am System (meist durch die *Adaptive Maintenance*, aber auch durch *Preventive Maintenance*) neue Fehler in das System eingebaut werden. Diese Fehler müssen, ungeachtet des eventuellen Verlustes an Reputation, Strafzahlungen und weiteren Unkosten, funktional beseitigt werden und das kostet wieder einiges an *DevC* sowie *TtM*. Man könnte auch hier die Investitionen pro *FP* zählen, jedoch ist es bei Fehlern schwer, die dazugehörige Anzahl an *FP* zu bestimmen, deswegen wäre eine Metrik an Investition pro Fehler geeigneter. Da Fehler aber unterschiedlichste Ausmaße annehmen können, ist dies auch nicht gerade ein einfacher Weg und bedarf einiger Vorüberlegungen. Als einzige Kategorie setzt die *Corrective Maintenance* keine neuen Features oder Anforderungen um, weswegen man davon ausgehen kann, dass keine Fehler durch neue Funktionalität oder neue Struktur mehr hinzu kommen, was zwar in der Praxis durchaus der Fall sein kann, aber theoretisch nicht mit dem höheren Fehleraufkommen durch *Adaptive Maintenance* und *Preventive Maintenance* gleichzusetzen ist. Die dritte Kategorie, die *Preventive Maintenance*, beinhaltet das Refactoring und Rearchitecting von Teilen der Software. Dies kann entweder durch geänderte Anforderungen, Ideale, Techniken oder Methoden geschehen, oder durch eine Säuberung und Vereinfachung des Systems. Alle diese Szenarien sind hier wieder in Investitionen pro *FP* messbar, was einen Richtwert darstellt, inwieweit anpassbar das System auf neue Gegebenheiten ist. Da hierbei große Änderungen am System bevorstehen, sind Fehler und damit Einfluss auf *Corrective Maintenance* nicht auszuschließen. Ebenso kann häufige *Adaptive Maintenance* eine *Preventive Maintenance* nach sich ziehen, damit die *Agility* und *Resilience* entsprechend hoch gehalten werden.

Hierbei stellt sich am Ende die Frage, inwieweit der strukturelle Wert den Geschäftsnutzen in der Einschätzung über das System ablöst oder zumindest beeinflusst. Wie in Kapitel 1.2 „Motivation für den Nachweis von Werten der Architektur“ erläutert, ist es schwer, den strukturellen Wert auf wirtschaftlicher Ebene einzubeziehen, da bislang nur grob gefasste Richtlinien und Frameworks die Architektur zu dem strukturellen Wert handhaben. Aber es ist einfach zu erkennen, dass eine schlechte Struktur hohe Kosten in den drei genannten Kategorien hervorruft, welche am Ende die Rechnung über den *NPV* mehr als erheblich beeinflusst. Die Frage ist dann immer, inwieweit ist der Schaden an der Struktur am System so hoch, dass ein System trotz gutem *NPV* wirtschaftlich schlecht dasteht. Es ist dahingehend die Aufgabe eines Architekten, das System so zu konstruieren, dass der Fall der schlechten Struktur gar nicht erst eintritt.

2.2 Grundlagen von Softwarearchitektur

Der Begriff Softwarearchitektur ist sehr weit gefasst und umfasst viele einzelne Themenbereiche. Man findet die Einflüsse der Softwarearchitektur in nahezu allen Entwicklungs- und Lebensphasen eines Softwaresystems, angefangen von der Spezifikation der Anforderungen bis hin zur Abnahme von System- und Akzeptanztests. Doch auch danach muss oft noch auf die Architektur zurückgegriffen werden, wenn die Software einer geregelten Evolution unterliegen soll. Dabei ist es wichtig, dass der dafür verantwortliche Architekt der Software seinen Einfluss für eine geordne-

te Entwicklung des Systems nutzt, um die geforderten Qualitätseigenschaften (beziehungsweise die nicht-funktionalen Anforderungen) umzusetzen. Die folgenden Abschnitte sollen das große Gebiet der Architektur etwas umranden und anschließend auf die Kernkonzepte eingrenzen, welche für diese Arbeit benötigt werden. Dabei werden durchaus Gebiete der Softwarearchitektur nur angeschnitten oder ausgelassen, die jedoch bei Interesse an geeigneteren Stellen nachgeschlagen werden können [3] [22] [23].

2.2.1 Architektur in Softwaresystemen

Softwarearchitektur kann sowohl als Aktivität als auch als Resultat der Aktivität genannt werden. Die Aktivität bezeichnet die Arbeit der Softwarearchitekten und das Resultat ist eine fertige Struktur der Software. Es ist wichtig zu verstehen, dass jedes Softwaresystem eine Architektur besitzt, der Unterschied liegt hierbei entweder in der impliziten oder expliziten Struktur. Bei der impliziten Architektur wird diese von selbst entwickelt und es gibt keinen Weg, diese zu kontrollieren. Es ist die Aufgabe der Softwarearchitekten, eine geeignete explizite Architektur für das System zu konstruieren [41], damit diese anschließend kontrolliert und in entsprechende Bahnen gelenkt werden kann.

Es gibt viele verschiedene Definitionen von Softwarearchitektur, aber die beiden verbreiteteren Definitionen sind folgende:

1. *„The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.“* [26], S. 3.

2. *„The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.“* [3], S. 6.

Die erste Definition ist dahingehend sehr beliebt, da sie den Wesensgehalt von Softwarearchitektur gut umfasst. Besonders die Betonung auf die *Principles* und ihrem Design sowie Evolution ist hier anzumerken. Diese Definition ist allgemeingültig als ISO/IEEE standardisiert und dient als Grundlage für Furrer [19]. Die zweite Definition ist mit den beiden Wörtern *components* und *relationship* sehr abstrakt gehalten und kann damit auf die verschiedensten möglichen Szenarien bei der Softwarearchitektur angewendet werden: auf Klassen, Methoden, Packages, Laufzeitobjekte, Prozesse, Protokolle, Rechner sowie all deren Zusammenhänge [34]. Es ist die Aufgabe des Softwarearchitekten, eine geeignete Zuordnung der beiden Begriffe auf die aktuelle Situation abzubilden.

Aktuelle Softwarearchitektur beginnt gewöhnlich bereits bei der Spezifizierung von Anforderungen. Dazu zählen ebenfalls Anforderungen von Qualitätseigenschaften, welche durchaus kritisch für das Unternehmen sein können. Dazu zählen Eigenschaften wie Verfügbarkeit, Wartbarkeit, Erweiterbarkeit, Performance, Sicherheit, Robustheit, Benutzbarkeit, Skalierbarkeit und viele weitere. Das Management und besonders die Zusicherung solcher Eigenschaften bedarf der Mitwirkung von Architektur in allen Phasen der Entwicklung von Software, von Design- bis Test-Spezifikationen. Für die meisten dieser Qualitätseigenschaften gibt es Metriken, welche kontinuierlich in mehreren Phasen der Entwicklung überprüft werden können. Dazu im Kapitel 3.4 „Metriken in Softwaresystemen“ mehr.

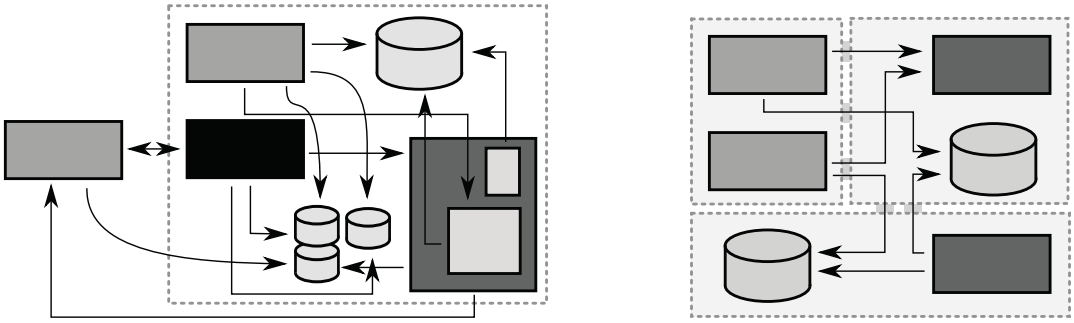
2.2.2 Einfluss und Auswirkung von Architektur

Das Resultat des wirtschaftlichen Verlusts ist wohl der deutlichste Hinweis darauf, warum ein Mehraufwand in die Architektur einer Software gesteckt werden sollte. Es soll in diesem Abschnitt noch einmal in groben Zügen aufgezeigt werden, wo genau der Einfluss von Architektur während der Entwicklungsphase sowie im Maintenance der Software zum Tragen kommt. Dabei spielt hier der Begriff „Zukunftsfähig“ eine übergeordnete Rolle, da, wie bereits in den oberen Kapiteln angerissen, dies ein bedeutender Faktor auf die Wirtschaftlichkeit eines Systems darstellt.

Der Einfluss von Architektur auf den Prozess der Softwareentwicklung ist in den vergangenen Jahren erheblich angestiegen. War zu Beginn von Softwarearchitektur das vorherrschende Thema *Ports und Konnektoren von Softwarekomponenten*, gibt es derzeit bereits neue Probleme in der Art und Weise von Architektur, die mit diesem Schema nicht gedeckt sind. Diese Probleme entstanden durch eine Weiterentwicklung von Architektur hin zu einer Thematik, welche viele Bereiche auch außerhalb des Designs von Software erfasst. Dabei verbindet Architektur mittlerweile weite Teile der *Application Software*, den *Infrastructure Services*, der *Technical Infrastructure* sowie den *Commodities Sourcing* [19]. Dabei passte sich die Architektur der Entwicklung von Systemen an, welche ihrerseits in *Complexity*, *Criticality* und *Rate of Change* stark zunahmen. Die Architektur wurde und wird durch den zunehmenden Einfluss von Software auf Unternehmen begründet. Durch diese Art von Zunahme dieser Thematik musste sich die Softwarearchitektur dahingehend weiterentwickeln, um auch Probleme von Software neuerer Generation handhaben zu können.

Eine Vernachlässigung von Architektur und das Management der Softwareevolution wirken sich garantiert in Zukunft negativ aus [19]. Die dafür verantwortlichen Phänomene der *Architecture Erosion* sowie der *Technical Debt* werden beide noch einmal genauer in dem Kapitel 2.3 „Fehler durch Architektur und Management“ erläutert. Die *Architecture Erosion* geschieht von allein und ist den äußeren Einflüssen geschuldet, zum Beispiel durch veraltete Techniken, neue Designs oder andere Ausgabekanäle. Die *Technical Debt* wurde bereits in den vorherigen Kapiteln erwähnt und entsteht hauptsächlich durch innere Einflüsse bei der Entwicklung oder Maintenance eines Systems, zum Beispiel durch Workarounds oder fehlerhafter Einbindung von Komponenten. Beides führt zu einer Herabstufung des Wertes des Systems in Bezug auf die Zukunftsfähigkeit sowie einer erschwerten Weiterentwicklung des Systems.

Ein System, welches mehrere Jahre über benutzt und erweitert wurde (ein *Legacy System*) und gleichzeitig den negativen Einflüssen von *Architecture Erosion* oder *Technical Debt* unterlag, endet später als das mehrmals angesprochene *Fossil System*, der Zustand, indem das System kaum noch wartbar ist und auch Erweiterungen sich wirtschaftlich absolut nicht mehr lohnen. Dieses System ist dann nur noch unter hohem Risiko (geringe *Resilience*) und ohne Erweiterbarkeit (geringe *Agility*) verfügbar, was wiederum enorme wirtschaftliche Folgen für das Unternehmen haben kann und haben wird. Spätestens in diesem Zustand des Systems stehen nur noch die Optionen des umfangreichen Refactoring, also dem teilweisen Ersetzen ganzer Teile des Systems, oder der Neuanschaffung eines anderen Systems zur Verfügung. Deshalb ist es wichtig, auch im Bezug auf die Zukunftsfähigkeit eines Systems, bei allem Streben nach höherem *Business Value* die anderen Werte *Agility* und *Resilience* nicht zu vergessen und zu pflegen, sowie gleichzeitig permanent der *Architecture Erosion* und dem *Technical Debt* entgegenzuwirken. Um den Vergleich zu der Gebäude-Architektur aus dem vorherigen Abschnitt erneut aufzugreifen, ist eine sehr vereinfachte Veranschaulichung in Abbildung 2.2 gegeben. Man kann erkennen, dass eine Vernachlässigung von Architektur und Entwurf in einem Zustand endet, welcher zu komplex und nicht mehr nutzbar wird (Abbildung 2.2a). Dagegen wird durch definierte Schnittstellen, simpler



(a) Ungenügende Architektur: erhöhte Komplexität und Erosion[1]

(b) Gute Architektur: erhöhte Stabilität und Zukunftsfähigkeit[2]

Abbildung 2.2: Auswirkungen vernachlässigter Architektur bei Software

Struktur, einheitlicher Semantiken und sinnvoller Trennung ein Systemstand erreicht, der Jahre und Jahrzehnte überdauern kann, da die Erweiterbarkeit (*Agility*) und Resistenz (*Resilience*) erhalten bleibt (Abbildung 2.2b).

Durch eine gute Architektur und damit verbundenen Kontrolle der positiven Entwicklung der Qualitätseigenschaften ist der Grundstein für ein gutes und zukunftssicheres Softwaresystem zugesichert, welches dann auch als „Investitionsgut“ verstanden werden kann. Es ist eindeutig, dass es nicht nur Architekten bei dem Bau von großer Software erfordert, sondern ebenfalls gute Anforderungs-Analytiker, Designer, Programmierer, Qualitätsmanager und Tester, Projektplaner, Vermarkter, System-Techniker und viele weitere Experten in diversen Bereiche benötigt, welche essenziell an großen Softwaresystemen beteiligt sind. Es ist jedoch die Architektur, welche in jedem der Bereiche faktisch interdisziplinär beachtet und umgesetzt werden muss, damit das Gesamtkonzept der Software einen positiven Ausgang findet und *Architecture Erosion* sowie *Technical Debt* an keinem Punkt der Entwicklung einen Einzug finden.

2.2.3 Architekturprinzipien

Um die in den vorherigen Kapiteln erwähnten negativen Folgen von schlechter Architektur abzuwenden, können sich damit betraute Architekten sich aus einer Anzahl an Mechanismen und Werkzeugen bedienen, welche größtenteils den geregelten Aufbau sowie anschließende Entwicklung des Systems sicherstellen. Dazu zählen *Methoden*, wie Prozesskoordination, Separierung von Aufgaben, Delegation und Beratung durch Experten, Team- und Stakeholder-Kommunikation, Consulting in verschiedenen Bereichen der Softwareentwicklung und weitere. Ebenso kann auf *Werkzeuge* zurückgegriffen werden, welche die Architektur-Arbeit unterstützen können. Dazu zählen Architektur-Programme, Architektur-Beschreibungssprachen, interne Workshops, Architektur-Frameworks, Patterns, Guidelines oder auch definierte Architekturprinzipien. In dieser Arbeit soll es vor allem um den letzten Punkt, die *Architecture Principles*, gehen. Diese sind ein bewährter Grundbaustein, welcher als Werkzeug für Architekten geeignet ist, um solide und zukunftsfähige Systeme zu entwerfen.

Architekturprinzipien sind dabei selbst ein weit gefächerter Bereich. Grundsätzlich können diese wie folgt definiert werden:
„Architecture Principles are fundamental insights – formulated as rules – how a good software system should be build.“ [19], Satz 3.
Die Prinzipien können, wie in der Definition beschrieben, als Regeln verstanden werden. Dabei

stellen diese Regeln bei Beachtung sicher, dass zumindest der Teil des Systems, auf den sich diese Regel bezieht, sich gut entwickelt. Dabei können diese Prinzipien in der Entwurfsphase der Entwicklung festgelegt werden, müssen jedoch in allen weiteren Phasen auf Einhaltung überprüft werden. Inwieweit ein Prinzip dem späteren System nutzt, muss individuell entschieden werden, denn wie in allen Disziplinen kann auch hier viel Overhead in deren Entwicklung und Umsetzung gesteckt werden. Auch die Anzahl der benutzten Prinzipien kann stark variieren, je nachdem, was umgesetzt werden soll. Dabei ist es wichtig zu beachten, dass jedes eingeführte Architekturprinzip einen Mehraufwand und höhere Investitionen erfordert, um es korrekt umzusetzen, weswegen auf einen soliden Mittelweg bei der Auswahl der Prinzipien geachtet werden sollte. Für die Qualitätseigenschaft der *Agility* existieren derzeit 12 etablierte Architekturprinzipien [19], welche umgesetzt werden können. Zusätzlich dazu verfügen weitere Qualitätseigenschaften über eigene Prinzipien, welche ebenfalls in großen Projekten umgesetzt werden sollten.

Die Ausmaße der entsprechenden Prinzipien können verschiedene Detailstufen aufweisen. Ein Beispiel für ein sehr grob gefasstes Architekturprinzip wäre folgendes: „Bei dem Wiederverwenden von Komponenten in einem System sollte prinzipiell die Black-Box-Technik genutzt werden und nur über Parametrisierung sowie Konfigurationen steuerbar sein.“ [19] Diese Regel ist so allgemein gefasst, dass sie fast das komplette System betrifft, sollte man mit Wiederverwendung (*Re-use*) arbeiten. Gleichzeitig ist sie so generisch, dass man sie nicht direkt auf das System umsetzen kann. Es erfordert eine Analyse des zukünftigen Systems, um erkenntlich zu machen, wo genau diese Regel ihren Einfluss findet, sodass am Ende des Softwareentwurfs das gesamte Prinzip immer noch gilt. Ein Beispiel für ein sehr konkretes Prinzip wäre dagegen folgendes: „Der Zugang zu dem System ist über Authentifizierung und Autorisierung eindeutig geregelt.“ [23] Dieses Prinzip ist bereits viel spezieller und es benötigt nur geringen Aufwand, die Komponenten im System auszumachen, welche den Login-Vorgang sowie Zugriffe regeln. Dabei besitzt dieses Prinzip einen viel engeren Bezug zu der Funktionalität des späteren fertigen Systems. Trotz alledem besitzen beide Sätze ihre Berechtigung als Architekturprinzip, da beide der Definition entsprechen: sie fungieren als grundlegendes Element, als „ewige Wahrheit“, welche das System zukunftsfähiger und wertvoller werden lassen, solange sie eingehalten werden.

In Greefhorst *et al.* [23] werden *Architecture Principles* als Unterkategorie der allgemeinen Prinzipien verstanden. Sie unterscheiden diese in die beiden folgenden Kategorien:

1. *Scientific Principles* sind solche, welche als „Kern des Engineerings“ bezeichnet werden. Hierbei spielen hauptsächlich Naturgesetze eine Rolle, welche für diverse Bereiche des Engineerings wichtig sind. Diese Prinzipien (zum Beispiel das „Gesetz von Gravitation“) sind gegeben und unausweichlich, haben aber direkten Einfluss auf das Ergebnis eines Projektes (zum Beispiel im Flugzeugbau).

2. *Design Principles* sind solche, welche durch vorherrschende Normen definiert werden und in Bezug auf das Entwerfen von Entwicklungs-Artefakten die Freiheiten des Designs einschränken. Damit wird eine geregelte und gute Software gewährleistest. Im Gegensatz zu den *Scientific Principles* sind die *Design Principles* durchaus vernachlässigbar, wenn sie nicht korrekt durchgesetzt werden, was negative Folgen im Resultat nach sich zieht.

Eine besondere Gruppe der oben definierten *Design Principles* stellen nach Ansicht von Greefhorst *et al.* [23] dann die erwähnten *Architecture Principles* dar, eingegrenzt auf die konkrete Entwicklung von Architektur-Artefakten und deren Umsetzung. Sie regeln das generelle oder spezielle Verhalten von Eigenschaften, die das zukünftige System beinhalten sollte. Da entsprechende Regelungen auch in mehreren Phasen der Entwicklung beachtet und umgesetzt

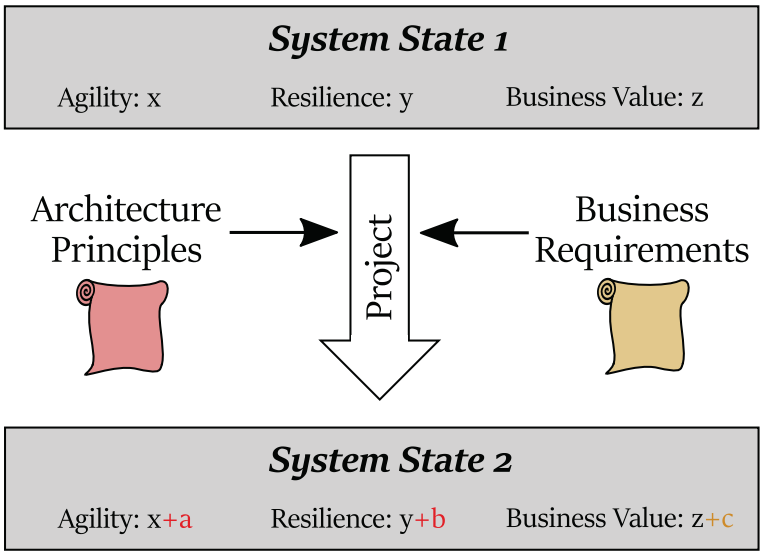


Abbildung 2.3: Der Einfluss auf ein Projekt

werden müssen, sind Softwarearchitekten in allen davon betroffenen Stellen involviert. In Abbildung 2.3 ist schematisch dargestellt, inwieweit neben den normalen Requirements auch die Architekturprinzipien eine Rolle spielen. Beides muss explizit auf das neue System oder die neue Erweiterung angewendet werden, um am Ende eine Steigerung der Zukunftsfähigkeit beeinflussenden Werte *Agility*, *Resilience* und *Business Value* zu erhalten (siehe dazu auch Kapitel 2.4 „Zukunftsfähige Softwaresysteme“). Demzufolge kann man die Architekturprinzipien als permanente interne Anforderungen an das neue System oder die neue Erweiterung ansehen, welche in einem gut organisierten Unternehmen auch voneinander separat niedergeschrieben, gehandhabt, eingearbeitet und überprüft werden, jedoch ständig eine Abwägung beider vorgenommen wird, um ein *Overengineering* (durch niedrige Priorität von *Business Requirements*) oder *Fossil System* (durch niedrige Priorität von Architekturprinzipien) zu vermeiden, dargestellt in Abbildung 2.4. Es besteht daher die Notwendigkeit, alle Argumente abzuwägen und zum Beispiel die richtige Art und Anzahl an umzusetzenden Prinzipien für die notwendigen Systemanforderungen zu finden.

Im Rahmen dieser Arbeit wurde genau eines dieser Prinzipien ausgewählt, das der „Redundanz-Freiheit“. Was genau dieses im Detail besagt, wie es interpretiert werden kann, wie es umgesetzt und gemessen werden kann, folgt in dem darauf ausgelegten Kapitel 4 „Redundanz“. Es sei hier bereits erwähnt, dass dieses Prinzip regelt, inwieweit Redundanz in einem System gehandhabt werden sollte, damit die drei Schadenskategorien *Complexity*, *Change* und *Uncertainty* sowie die damit verbundenen Kostenkategorien *Adaptive Maintenance*, *Corrective Maintenance* und *Preventive Maintenance* minimiert werden, was sich dann wiederum positiv auf die Zukunftsfähigkeit auswirkt. Das genau dieses Prinzip gewählt wurde, liegt durchaus auch an der großen Bedeutung für heutige Unternehmen, da unkontrollierte Redundanz im System einer der größten Komplexitäts-Treiber und Fehlerursachen ist. An sich ist die Kontrolle der Redundanz nur ein Prinzip von vielen, das auf ein System angewendet werden kann, und es muss in jedem System neu entschieden werden, ob dies zum Repertoire der gewählten Prinzipien dazu zählt, was sehr oft der Fall ist.

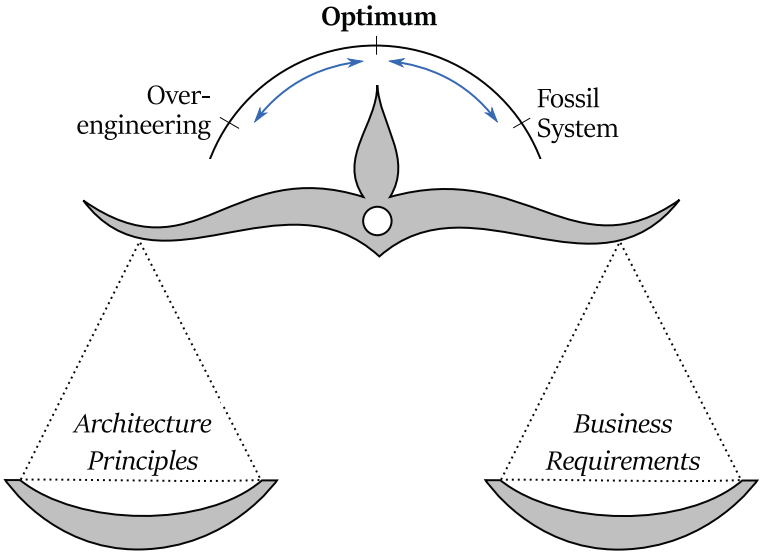


Abbildung 2.4: Die Balance zwischen *Business Requirements* und *Architecture Principles*

2.3 Fehler durch Architektur und Management

Softwaresysteme definieren sich über viele verschiedene Bereiche. In jedem davon kommt es auf die Qualität der Planung und Umsetzung an. Einer dieser Bereiche ist die Architektur und deren Planung. Eine gute Architektur kann Softwaresysteme hervorbringen, welche Jahrzehnte überdauern, in vielen Gesichtspunkten anpassbar bleiben und die Fähigkeiten eines Unternehmens enorm steigern. Die Kehrseite ist schlechte oder implizite Architektur, welche kostenintensive Systeme in allen Bereichen hinterlässt, nur marginal an Eigenwert besitzt und damit kaum Unternehmenswert generiert.

Gute Architektur wird meist durch einen dedizierten Architekten oder einer dafür zuständigen Person (respektive eines Teams) konstruktiv in den Entwicklungs- und Evolutionsphasen aufgebaut. Dabei spielen natürlich seine Erfahrung sowie Kenntnisse in den Bereichen der Softwarearchitektur eine Rolle, genauso sowie seine zur Verfügung stehenden Werkzeuge. Eine Stelle, an denen ein Softwarearchitekt definiert wird, findet sich in Kapitel 2.2 „Grundlagen von Softwarearchitektur“. Viel Arbeit und einigermaßen hoher Aufwand sollte in die Definition und Umsetzung von Architektur fließen, um am Ende des Prozesses ein Softwareprodukt zu erhalten, welches auch als „Investitionsgut“ bezeichnet werden kann. Im Gegensatz dazu ist die Anwendung von schlechter Architektur ein nur sehr geringer Aufwand. Im Grunde benötigt man kein eigenes Zutun, um den Prozess der *Architecture Erosion* vollziehen zu lassen. Dieser endet in dem schon einige Male angesprochenem *Fossil System*. Das beste Beispiel ist hier ein Team von Entwicklern, welche jeweils unterschiedliche Komponenten schreiben, wovon jeder seine eigene Vorstellung der Gesamtarchitektur sieht. Dieses unorganisierte dezentrale Architektur-Management kann durchaus einer der Hauptgründe sein, warum ein System an Wert verliert und an Kosten zunimmt.

Häufig ist es auch nicht die bewusste Vernachlässigung von Architektur und deren Management, welche das System an Wert verlieren lassen. Oft sind es einfach auch unbewusste Elemente oder Entscheidungen im Entwicklungsprozess. Der Artikel [67] macht deutlich, wie einfache, uneinheitliche Vorgehensweisen innerhalb der Architektur und dem Design von Software in einen Verlust von mehreren Millionen Dollar führen können. Eines der prominentesten Beispiele, die

in diesem Artikel genannt wurden, ist der Verlust der 1998 gebauten Rakete Delta III. Diese baute auf ihrem Vorgänger Delta II auf, jedoch wurden einige Änderungen in den Hardware-Steuerwerken integriert. Die darauf beruhende Software stammte noch aus Delta II und wurde, weil bereits schon einmal getestet, einfach in Delta III übernommen. Der Ausgang war, dass wegen fehlerhafter Integrationsarchitektur der alten Software auf die neue Umgebung (Steuerwerke) die entsprechenden Triebwerke nicht wie gewollt funktionierten und die Rakete gesprengt werden musste. Das Ärgerliche daran war, dass dieser Fehler durch besseres Management hätte entdeckt werden können. Der Artikel besitzt gegen Ende einen Absatz, der hier in Bezug auf das Thema dieser Arbeit noch einmal zitiert werden soll:
„Genau wie bei der Ariane 5 führte die Übernahme der Software von einer alten Rakete in ein neues System zum Verlust einer Mission. In beiden Fällen gab es die Überzeugung, dass die alte Software die Fehlerfreiheit garantieren würde. Aber tatsächlich wurde sie zur Ursache eines völlig neuen Problems.“ [67], S. 2, Abs. 7.
Die genannten Beispiele sind zwar sehr drastisch, jedoch ebenso mit einem langsamen Verfall von Softwaresystemen vergleichbar, genau dann, wenn die Auswirkungen nicht sofort spürbar werden, wie in den genannten Beispielen, sondern über einen längeren Zeitraum ihre Kosten einfordern.

Technical Debt Die *Technical Debt* [58] ist das Phänomen, das ein System von innen heraus zerfallen lässt. Es ist die Art von Schaden, welche immer dann entsteht, wenn man etwas anderes als die eigentliche mögliche Ideallösung einbaut. Das beinhaltet das Umgehen von definierten Interfaces oder auch das Ignorieren von definierten Typen für bestimmte Elemente im System. Dieser Schaden muss (später) durch Mehraufwand beseitigt werden. Dabei ist die Messung des verursachten Schadens sehr schwierig, da die *Technical Debt* eine Vielzahl abstrakter Fälle abdeckt. Den Begriff der *Debt* im System wurde 1993 im Report Cunningham [11] erstmalig in Umlauf gebracht, von wo aus sich die *Technical Debt* als reales Problem etablierte und aufgenommen wurde. Wie in Brown *et al.* [4] geschildert, kann man die *Technical Debt* direkt in eine „Schuld“ übersetzen, die dadurch generiert wird, dass bewusst oder unbewusst diverse Abkürzungen oder Abweichungen bei der Entwicklung des Systems vorgenommen werden. Solange diese Schuld nicht getilgt wird, häuft sich der Schaden durch die Schuld in Zukunft weiter an und verursacht immer mehr Probleme in der weiteren Evolution des Systems. Nach den Untersuchungen von Klinger *et al.* [29] ist häufig die Abwesenheit eines informierten Architekten und die direkte Einwirkung von anderen Stakeholdern der Grund für neue *Technical Debt* im großen Systemen.

Der Klassiker an bewusster *Technical Debt* ist hierbei der Satz „Ich habe gerade keine Zeit dafür, dieses Feature sauber umzusetzen. Das erledige ich später.“, was anschaulich das Aufnehmen von *Technical Debt* bedeutet. Diese kleinen Workarounds bei der Implementierung, aber auch bereits beim Design von Software führen zu unnötiger Komplexität, Fehlerfälligkeit, schlechter Wartbarkeit und Weiterentwicklung, was wiederum die Kosten in den Bereichen *Evolution*, *Refactoring* und *Operativ* steigert. In den meisten Fällen bewusster Vernachlässigung von Architektur geschieht dies aufgrund mangelnder Zeit, knapper Deadlines oder kleinen Budgets. Aber auch das System selbst kann durch hohen Testaufwand oder „widerspenstiger“ Technik daran Schuld tragen [34]. Das kann alles durchaus seine Richtigkeit haben, aber wie in Brown *et al.* [4] geschildert, ist es nötig, diese aufgebaute Schuld nächstmöglich wieder abzubauen. In Lilienthal [34] sind die der Autorin aus der Praxis bekanntesten Gründe kurz aufgeführt: das Phänomen „Programmieren kann jeder“, unbemerkte Architekturerosion, Komplexität und Größe, das Unverständnis von Management und Kunden für Individualsoftware.

Hierbei kann angemerkt werden, dass dies nicht die einzigen Ursachen sind, und auch die Ansicht, dass *Architecture Erosion* zu *Technical Debt* führt, muss man nicht vertreten. Jedoch kann man sich vorstellen, dass ab einer gewissen Größe und Komplexität das System verlockender für diverse Workarounds und Shortcuts wird, wenn es nicht entsprechend flexibel gestaltet wurde. Zudem spielen durchaus die Entwickler selbst eine Rolle (beziehungsweise deren Erfahrung) und die Akzeptanz der Kunden und des Managements für gute Qualitätssoftware.

Jedoch besitzt diese *Technical Debt*-Thematik ein ganz eigenen Problem: das Auffinden von bereits (eventuell seit langem) eingebauter *Technical Debt*. Laut aktuellen Schätzungen liegen die Kosten von *Technical Debt* im Schnitt bei 3.61 Dollar pro *LOC* und bei 1 Million Dollar pro Softwaresystem [5]. Es gibt bereits einiges an Methoden und Literatur, welche auf Code-Ebene diverse Stellen und Mechaniken angeben, wie solche Workarounds (beziehungsweise insgesamt *Technical Debt*) herausgefiltert werden können [12] [32] [44]. Häufig in allen Quellen genannt ist dabei die *Code Coverage* (der Anteil an Code-Segmenten, welche von Tests gedeckt sind), *Cyclomanic Complexity* (als Maß für eindeutige Abläufe in einem Programm), *Duplicate Code* (der Anteil an duplizierten Quellcode im Programm) sowie *Coupling and Cohesion* (Metrik für den Zusammenhang und Abhängigkeit von Komponenten). Auch wird sehr häufig eine Analyse von Variablenverwendung und -verhalten benutzt. Wobei einige Probleme, wie auch in den Beispielen weiter oben, mit solchen Werkzeugen hätten gefunden werden können, ist die *Technical Debt* und *Architecture Erosion* kein Problem auf reiner Code-Ebene. Hierbei können viele Faktoren einspielen, unter anderem falsche Design-Entscheidungen, falsches Layering (Schichtung) der Applikation, fehlerhafte Benutzung von Techniken oder auch Duplikate in Semantiken oder Anforderungen. Aber auch da gibt es erste Ansätze, die *Technical Debt* abseits der reinen Code-Ebene zu analysieren, wie es in Weber *et al.* [53] beispielsweise für *Technical Debt* in Datenbank-Schemata aufgezeigt wird. Ebenso wurde bereits in Nugroho *et al.* [42] ein empirisches Modell für *Technical Debt* und deren reale Kosten vorgestellt. In welcher Form die *Technical Debt* erscheint, dieser muss immer aktiv entgegengewirkt werden, damit das System frei davon bleibt und der Wert sowie *Agility* und *Resilience* gleich bleiben oder gar erhöht werden können.

Architecture Erosion Die *Architecture Erosion* kann man ebenfalls als stetigen Verfall der Struktur durch Nichtstun verstehen, mit dem Unterschied, dass der Zerfall nicht durch die beteiligten Personen am System verursacht wird sondern ganz von allein geschieht. Im Laufe der Zeit verändert sich die Umgebung und der Umgang mit dem System: Funktionen werden obsolet, das Nutzerverhalten ändert sich, Funktionalitäten werden ersetzt, Programmierstile ändern sich, Konkurrenten haben bessere Features, neuere Technik existiert bereits oder neue Ausgabekanäle werden eingeführt. Das alles führt dazu, dass ein einst gut gebautes System mit der Zeit zwar die Funktionalität beibehält, aber die aktuelle Architektur derartig abgestuft wird, dass ein umfangreiches Rearchitecting oder ein Austausch des Systems notwendig sein kann, um die erwartete *Agility*, *Resilience* und den Geschäftsnutzen (und damit den Wert des Systems) aufrechtzuerhalten. Ein Beispiel, wie *Architecture Erosion* in das System eines Unternehmens Einzug erhält, geben die folgenden Artikel [47] [21]. Zwar ist in dem Beispiel nicht direkt die *Agility* betroffen, die *Resilience* jedoch umso mehr, da die *SHA-1*-Methode eigentlich nicht mehr zeitgemäß und zudem unsicher geworden ist. Jedoch haben sich die Verantwortlichen der aufgeführten Systeme aus internen Gründen gegen eine Erneuerung und damit für die *Architecture Erosion* entschieden.

Dabei ist es wichtig zu verstehen, dass die *Architecture Erosion* durch äußere Einflüsse auf das System bedingt ist. Dagegen ist die *Technical Debt* ein Verfall des Systems durch inne-

re Einflüsse, eingebaut durch Workarounds, nicht beseitigte Fehler, „Dead Code“, Duplikate, Nichtbeachtung von Richtlinien und anderes. Wie auch die *Technical Debt* muss die *Architecture Erosion* aktiv beseitigt werden, um keinen Schaden im Laufe der Zeit davonzutragen. Diese Unterscheidung ist in der Literatur nicht immer sauber und konsistent getrennt, wie zum Beispiel Lilienthal [34] die *Architecture Erosion* mit der *Technical Debt* relativ oft durchmischt. Das liegt auch mitunter an den nicht einheitlichen Definitionen. Es ist dennoch nachvollziehbarer, wenn die oben genannte Trennung beider Aspekte wie in Furrer [19] beibehalten wird, um den wirklichen Eigenanteil des Verfalls zu umranden.

Auswirkungen der Kostenkategorien Bleibt noch die Frage zu klären, was passiert, wenn durch andere Ursachen die drei Kostenbereiche der *Adaptive*, *Corrective* und *Preventive Maintenance* in die Höhe getrieben werden.

Sollte die *Adaptive Maintenance* bereits soweit angestiegen sein, dass jede Änderung am System zu einem großen Aufwand wird und das System selbst keinerlei Anzeichen von Zukunftsfähigkeit mehr ausweist, dann steigen die monetären und zeitlichen Mehrkosten für neue Erweiterungen des Systems ebenfalls stark an. Jede Erweiterung verursacht dahingehend mehr Aufwand, dass unnötige Abhängigkeiten, Redundanzen, Komplexität und andere negative Eigenschaften beachtet werden müssen. Dies macht zum einen nicht nur jede Erweiterung des Systems unattraktiv für Entwickler sondern kann auch soweit gehen, dass spezielle Anforderungen nicht mehr umgesetzt werden können. Dies besitzt einen erheblichen Einfluss auf das Unternehmen, da ein stagnierendes System als Kerngeschäft nicht konkurrenzfähig sein kann.

Die *Corrective Maintenance* betrifft nicht direkt die Erweiterbarkeit des Unternehmens, sondern eher die Folgen davon. Es verursacht stark anwachsende *TtM* und *DevC*, wenn bei jeder Erweiterung (beziehungsweise Änderung) des Systems unzählige Fehler auftauchen, die behandelt werden müssen. Dies kann im Endeffekt sogar die initialen Kosten der eigentlichen Erweiterung übertreffen, wenn einfach zu viel *Technical Debt* im System beinhaltet ist. Auch hier kann sich dadurch die Bereitschaft der Stakeholder für Änderungen am System verringern, was wiederum schädlich für das Unternehmen ist, denn ein gut funktionierendes System sollte immer einfach und unkompliziert neue Anforderungen umsetzen können. Diese „Zerbrechlichkeit“ des Systems bei Änderungen führt nun dazu, dass gewisse Bereiche des Systems stetig mit Mehraufwand belegt sind, da sich (eventuell sogar dedizierte) Entwickler nur auf das Fixen der Fehler aus dem alten System konzentrieren müssen, falls das noch möglich ist.

Schlussendlich führt eine erhöhte *Preventive Maintenance* dazu, dass ein einmal mit *Technical Debt* und *Architecture Erosion* belegtes System dazu tendiert, den Aufwand zu erhöhen, den es benötigt, um alle Fehlentscheidungen zu bereinigen. Hierzu sei angemerkt, dass es dabei nicht um Fehlerbehebung geht, sondern um das Umstrukturieren des Systems und das proaktive Bereinigen von (auch im Nachhinein) falschen Voraussetzungen und Entscheidungen. Dabei kann man davon ausgehen, dass dieser Mehraufwand zum kompletten Bereinigen aller schadhaften Teile im System nicht unbedingt linear ansteigen muss und durchaus schnell einen exponentiellen Verlauf annehmen kann. Dadurch ist es (abgesehen von überhaupt keine Fehlentscheidung zu treffen) immer am besten, Fehlentscheidungen und strukturelle Änderungen am System schnellstmöglich zu korrigieren. Sollte dies ausbleiben, dann hat man neben den bereits existierenden Kosten aus *Adaptive* und *Corrective Maintenance* auch damit zu kämpfen, dass neue Anforderungen nicht so umgesetzt werden können, wie es am einfachsten wäre, sondern wie es das System erzwingt, auch wenn man die Ideallösung kennen würde. Die *Preventive Maintenance* muss stetig betrieben werden, allein schon aus der passiven Wirkung der *Architecture Erosion* auf das System. Das bedeutet, man wird um einen Mehraufwand durch *Preventive Maintenance* nicht herum

kommen, man kann diesen Betrag allerdings sehr klein halten, wenn man eine gute Architektur und Entwicklungsprozesse im Unternehmen verfolgt. Wird dies vernachlässigt, wird das System zwangsläufig in einen nicht mehr reparierbaren und erweiterbaren Zustand verfallen, sodass eine kostspielige Neuanschaffung aufgezwungen wird, wenn man auf das System als Unternehmen angewiesen ist.

2.4 Zukunftsfähige Softwaresysteme

Software ist fester Bestandteil der Zukunft in unserer Gesellschaft, was bedeutet, dass diese Software ebenfalls für die Zukunft ausgelegt sein sollte. Dieses Kapitel soll diese Thematik der Zukunftsfähigkeit etwas veranschaulichen und die einzelnen Aspekte der Zukunftsfähigkeit einzeln kurz ansprechen und deren Bezug zur Architektur darlegen.

2.4.1 Langlebige Software

Langlebige Software ist eine essenzielle Eigenschaft von Software, damit diese sich auf dem Markt etablieren kann. Die Langlebigkeit ist dabei selbst nur eine Unterkategorie der Zukunftsfähigkeit einer Software, jedoch eine der wichtigeren. Die Grundlage für diesen Abschnitt bildet Lilienthal [34], worin die wichtigsten Methoden, Aspekte, Modelle und Architekturen für langlebige Software auf Code-Basis beschrieben werden.

Ein wichtiger Aspekt von langlebiger Software sind die vier Eigenschaften, an denen man eine nicht-langlebige Struktur in Software erkennt. Diese hier genannten sind aus Martin [37] entnommen und beschreiben sie Symptome von „Rotting Design“, also im groben Sinne die Auswirkungen von *Technical Debt* und *Architecture Erosion*. Diese vier Eigenschaften sind die Folgenden:

1. *Rigidity* (Starrheit) beschreibt den Zustand des Systems, in dem eine Änderung im System viele weitere zwangsweise nach sich zieht. Eine Änderung an einer Komponente führt zu Abhängigkeiten, welche Änderungen an anderen Komponenten erfordern. Häufig können sich Entwickler auch in Unklarheiten verlieren.

2. *Fragility* (Zerbrechlichkeit) ist der Zustand, in dem eine Änderung neue Fehler hervorruft, welche mitunter nicht in direkten Zusammenhang mit der Änderung gebracht werden können. Das bedeutet auch, dass eine eigentliche Fehlerbehebung neue Fehler an anderer Stelle verursachen kann. Eine hohe *Fragility* ist dahingehend leicht anfällig für solche „spontanen“ Fehler. Ein gewisser Kontrollverlust über das System kann die Folge sein.

3. *Immobility* (Unbeweglichkeit) ist die Eigenschaft des Systems, dass im Hinblick auf neue Features keine Anpassbarkeit herrscht. Neue Elemente werden eher als „Copy-Paste“ eingefügt, anstatt alte Elemente wiederzuverwerten. Entweder, weil die alten Komponenten nicht auf Wiederverwendung ausgelegt waren oder zu viele Abhängigkeiten mitbringen. Das führt dazu, dass fast gleiche Funktionalität neu eingefügt wird, anstatt generisch wiederzuverwerten.

4. *Viscosity* (Zähigkeit) wird dadurch definiert, dass ein „Shortcut“ bei der Entwicklung (welcher als nicht Design-erhaltend betrachtet wird) schneller umgesetzt werden kann, als eine Design-erhaltende Lösung.

Gegen diese Auswirkungen müssen Architekten, Designer und Entwickler stetig ankämpfen [34], damit das System auf Dauer sinnvoll nutzbar und erweiterbar bleibt.

Weiterführend soll hier nochmal kurz aufgeführt werden, was eine Langlebigkeit bei Softwaresystemen ausmacht, ohne dabei ganz in die Details einzugehen, da viel davon bereits auf eine etwas andere Art in den folgenden Kapiteln erläutert wird, welche engeren Bezug zu dem Thema dieser Arbeit haben. In Lilienthal [34] werden verschiedene Konzepte vorgestellt, welche eine Software intuitiver wartbarer und in sich langlebiger macht. Diese Konzepte sind die Folgenden.

1. *Modularität* sorgt für eine flexible Handhabung des Systems. Modularisierte Systeme sind einfacher in der Wartung und Erweiterung, da die Unabhängigkeit der Einzelteile große Freiheiten gewährt und keine Abhängigkeiten verursacht. Dafür werden sinnvolle Metriken vorgeschlagen, wie *Cohesion*, *Size*, *Complexity*, und *Coupling*. Hierbei wird besonders auf das *Chunking* eingegangen, dem Kapseln und Zusammenfassen von Informationen in einzelne Einheiten, in der Softwareentwicklung unter anderem als *Information Hiding* bekannt. Für die entstehenden Module muss gelten, dass sie ein zusammenhängendes, kohärentes Ganzes bilden, welches nach außen nur explizite Schnittstellen aufweist und mit anderen Bausteinen nur minimal gekoppelt ist.

2. *Musterkonsistenz* bezeichnet die Handhabung der Entwurfsmuster und -Schemata im System und dessen Entwicklung. Dabei beschreibt die Soll-Architektur diverse Muster, an denen sich die Entwickler orientieren können. Es ist wichtig, dass alle am System beteiligten Personen sich auf eine Vorstellung von Entwurfsmustern einigen und diese umsetzen. Aber auch auf eine konsistente Umsetzung innerhalb einer Komponente muss auf eine konsistente Umsetzung von Mustern geachtet werden. Sind bestimmte Entwickler für bestimmte Komponenten verantwortlich, kommt es häufig dazu, dass diese ihre „eigene Note“ oder ihren „kreativen Anteil“ in den von ihnen entworfenen Code einbringen, da sie eine eigene Vorstellung der Umsetzung haben. Obwohl eine Eigenverwirklichung der Entwickler zu wünschen ist, muss dennoch auf sinnvolle und einheitliche Muster und Prinzipien wert gelegt werden, auch bei eigentlich voneinander unabhängigen Komponenten.

3. *Hierarchisierung* ist ein besonders wichtiges Konzept beim Verständnis von Systemen und deren Weiterentwicklung. Fast alle modernen Systeme sind in Schichten aufgebaut, welche über bestimmte Befugnisse verfügen, auf darunterliegende Schichten zuzugreifen. Aber auch hier müssen sich Architekten und Entwickler auf ein konsistentes Schichtsystem einigen. Die beiden Beispiele, welche in Lilienthal [34] genannt werden, sind die technische und fachliche Unterteilung des Systems. Während man die technische Schichtung auf „Präsentation“, „Applikation“, „Fachdomäne“, „Infrastruktur“ aufteilen kann, kann die fachliche Schichtung zum Beispiel in „Logging“, „Widget“, „Server“, „Client“ und „Mapper“ eingeteilt werden. Bei sehr großen Systemen lohnt sich die Überlegung, die technische und fachliche Schichtung zu vereinen und daraus eine Matrix zu erstellen, mit der Technik in der Vertikalen und der Fachdomäne in der Horizontalen. Egal wie man es am Ende umsetzt, es muss immer darauf geachtet werden, dass keine Schichtverletzung geschieht, da sonst die Vorteile einer Schichtung verloren gehen.

4. *Zyklenfreiheit* ist ebenfalls ein großer Abschnitt in Lilienthal [34]. Die Vorteile von Zyklen-freien Systemen liegen dabei auf der guten Erweiterbarkeit, Testbarkeit, Evolution, Partitionierung und Verständlichkeit. Bei allen fünf Eigenschaften würden sich Zyklen dahingehend auswirken, dass man einen Teil nicht unabhängig der am Zyklus beteiligten anderen Teile bearbeiten kann. Man kann damit die Zyklenfreiheit als Grundlage für gute Entwurfsmuster, sinnvolle Schichtung und Modularität ansehen, da nichts davon wirklich gut umsetzbar wäre, würden sich beteiligte Komponenten gegenseitig im Kreis bedingen.

Es ist in Lilienthal [34] genauestens erläutert, mit welchen Werkzeugen man automatische Analyse des Sourcecodes auf *Technical Debt* hin untersucht, um den Unterschied zwischen Ist- und Soll-Architektur aufzuzeigen. Weiterhin werden Beispiele und Konstruktionen von langlebigen Softwarearchitekturen aufgezeigt und die Differenzen bei verschiedenen Programmiersprachen ermittelt. Abgeschlossen wird mit Beispielprojekten aus der Praxis von der Autorin. Da all diese Themen jedoch über den Rahmen dieser Arbeit hinausgehen würden, sei hier nur weiterhin für Interessierte auf diese Literatur verwiesen.

2.4.2 Zukunftsfähige Architektur: Agility, Resilience und der Business Value

Ein zukunftsfähiges System ist in der heutigen Zeit der wichtigste Faktor in Unternehmen, welche stark auf Softwarelösungen setzen, wie zum Beispiel Banken oder Versicherungen, aber auch Großindustrie, Militär und ziviles Gewerbe wie Flughafen- und Bahnunternehmen. Sollte hierbei das System durch *Architecture Erosion* und *Technical Debt* in einen nicht mehr wartbaren Zustand verfallen, wäre damit nicht nur deren IT-Sparte bedroht, sondern meist das komplette Unternehmen.

Ein zukunftsfähiges System ist definiert nach den folgenden Gesichtspunkten:
„A future-proof software system is a structure that enables the management of complexity, change and uncertainty with the least effort, acceptable risk and specified quality properties.“ [19], Satz 1. In dieser Definition sind genau die gleichen Gesichtspunkte erkennbar, die auch bereits in den vorherigen Kapiteln angesprochen wurden: zum einen das Management von Struktur und Entwicklung über die *Non-Functional Properties*, beziehungsweise Qualitätsattribute wie Performance, Usability, Safety, und weitere, zum anderen der Einfluss auf den *Business Value* durch die Bereiche *Agility* („least effort“) und *Resilience* („acceptable risk“).

Die drei Bereiche in der Definition werden nicht umsonst so stark hervorgehoben, sind sie doch essenziell für ein stabiles System, gesichert für Entwicklungen in der Zukunft. Dabei werden alle drei Werte genauestens definiert, um den Fortschritt des eigenen Systems sichtbar machen zu können. Die hier beschriebenen Metriken stammen aus Furrer [19] und sind eine Möglichkeit, die entsprechende Zukunftsfähigkeit des Systems zu evaluieren.

1. *Business Value* ist im Endeffekt der Wert, welcher weiter oben als Wert für das Unternehmen zugesichert wurde. Per Definition generiert dieser Wert einen (wirtschaftlichen) Vorteil für das Unternehmen durch Vermarktung, aber auch durch Kosteneinsparung, Wettbewerbsvorteile (durch innovative Funktionalität), Prozessoptimierung, und weiteres. Dieser Wert ist deshalb wichtig, weil viele Projekte in der Planung einen positiven NPV vorweisen müssen, bevor diese umgesetzt werden und eventuelle Geldmittel gewährt werden. Da der *Business Value* bereits in Kapitel 2.1.3 „Der Wert eines Systems“ genauer erläutert wurde, wird hier nicht weiter darauf eingegangen. Es ist hierbei nur wichtig zu unterscheiden, dass dieser Wert kein „struktureller Wert“ der Software ist, sondern eine Kostenabschätzung aus der Projektplanung.

2. *Agility* ist der Wert für die Fähigkeit des Systems, sich an neue, unbekannte Anforderungen mit möglichst kleinen Aufwand umsetzen zu können. Das bedeutet insbesondere, dass diese neuen Anforderungen mit einer kurzen TtM sowie gleichzeitig geringen $DevC$ umgesetzt werden können. Dies ist deshalb in der heutigen Gesellschaft wichtig, da neue Technologien, Methoden und Anforderungen besonders schnell integriert werden müssen, um die Zukunftsfähigkeit zu erhalten. Insgesamt gibt der Wert für *Agility* ein Verhältnis wieder, inwieweit die Größe des Systems die beiden Ressourcen Zeit und Geld benötigt.

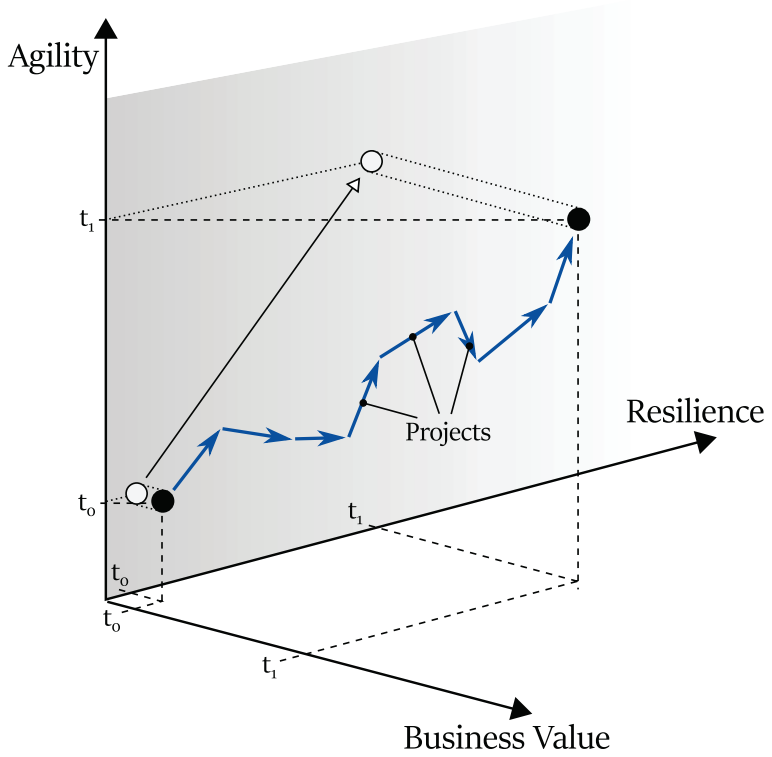


Abbildung 2.5: Das Koordinatensystem von *Agility*, *Resilience* und *Business Value*

Dabei ist die zugehörige Metrik, der *Agility Value*, ein reiner Vergleichswert und daher nur geeignet, um das System mit einem anderen, eigenen Stand oder gegen andere Systeme zu vergleichen.

3. *Resilience* ist der zweite wichtige Wert für ein erfolgreiches zukunftsfähiges System. *Resilience*, oder zu deutsch „Robustheit“, findet Anwendung bei Fehlern, Defekten oder Angriffen von dem oder gegen das System. Dies ist ebenfalls ein wichtiger Faktor, da ein robustes System das nötige Vertrauen und die Zuverlässigkeit schafft. Bei der Einschätzung für *Resilience* ist ein wichtiger Faktor, welchen Ausgang der Auslöser einer Störung verursacht. Kategorisiert wird dabei der *Crash* (der komplette Ausfalls des Systems), die *Degraded Operation* (das System ist zumindest zeitweise nur bedingt erreichbar oder nutzbar) sowie die *Malfunction* (das System verhält sich anders als gewünscht). Auch dieser Wert ist, wie auch *Agility*, ein reiner Vergleichswert, welcher genutzt werden kann, um das System gegen sich selbst oder andere zu evaluieren.

Diese Sichtweise mit den drei ausschlaggebenden Werten eines zukunftsfähigen Systems steht jedoch nicht im Gegensatz zur Eigenschaft der Langlebigkeit, sondern vielmehr verlaufen beide Eigenschaften bei der Evolution des Systems parallel nebeneinander her und müssen mehr oder weniger separat betrachtet werden. Während diese drei Eigenschaften eher im Sinne einer „Leitplanke der Entwicklung“ dienen, sichert die Langlebigkeit einer Software überhaupt die Möglichkeiten einer sinnvollen Struktur, welche für zukunftsfähige Systeme geeignet ist. Ein nicht-langlebiges System kann auch in keinster Weise auf guter *Agility* und *Resilience* aufbauen, und ein langlebiges System ohne diese beiden Attribute ist nicht evolutionsfähig.

Es ist sicher, dass ein System mit vielen guten Qualitätsattributen besetzt sein muss, um

wirtschaftlich zu sein. Jedoch sichert die *Agility* und *Resilience* die Evolution. Gerade weil so viele Module in der heutigen Welt eine lange Lebensspanne besitzen müssen, muss auch für deren Zukunft gesorgt sein. Eine hohe *Agility* sorgt für kostengünstige Evolution innerhalb kurzer Zeitspannen, effizientere Nutzung der Ressourcen des Unternehmens und ist damit ein wichtiger Faktor bei der Durchsetzung am Markt bei Konkurrenten. Die *Resilience* sichert die Funktionalität sowie Zuverlässigkeit und mindert das Risiko bei einfachen und kritischen Fehlern, Einbrüchen, Angriffen oder deren Folgekosten. Das hat ebenso Auswirkung auf die Marktdurchsetzung gegenüber Konkurrenten, da zum Beispiel nur die wenigsten Unternehmen ein System einsetzen wollen, das bei dem Großteil aller Eingaben abstürzt oder sensible Daten unverschlüsselt überträgt und damit unsicher ist. Dabei sind *Agility* und *Resilience* jedoch nicht nur zur Marktdurchsetzung geschaffen, sondern auch für das Unternehmen selbst, weil dadurch die einfache und sichere Handhabung und die Produktion des Systems sichergestellt ist, sodass auch bei für sich selbst produzierte Software beide Werte entsprechend hoch gehalten werden sollten. Insgesamt sorgt die *Agility* für den Erfolg des Systems, während die *Resilience* das „Überleben“ des Systems in der genutzten Umgebung sichert [19]. Die Vereinbarkeit dieser drei wichtigen Eigenschaften *Business Value*, *Resilience* und *Agility* ist heute essenziell bei der Entwicklung von großen Systemen. Eine bloße Verlagerung auf eine der drei Eigenschaften führt zu einer begrenzten Entwicklung des Systems in naher oder ferner Zukunft. Das Zusammenspiel ist in Abbildung 2.6 aufgezeigt, welches so in Furrer [19] genannt wurde. Man kann sehr gut erkennen, dass neben der fortwährenden Weiterentwicklung des *Business Value* ebenso die *Agility* und *Resilience* gesteigert wird. Dabei unterscheiden sich die Perspektiven der einzelnen, am System beteiligten Stakeholder nur durch eine spezifische Ansicht der Dimensionen. Während das Management des Unternehmens sich hauptsächlich auf die eindimensionale Achse des *Business Value* beschränkt, sollten Softwarearchitekten sich aller drei Dimensionen des Systems bewusst sein und besonders gegenüber dem Management auf die Entwicklung der Dimensionen *Agility* und *Resilience* einwirken. Inwieweit sich das System hinsichtlich der Zukunftsfähigkeit weiterentwickelt, wird durch die graue Ebene in der Abbildung dargestellt, welche durch die Achsen *Agility* und *Resilience* aufgespannt wird.

Nimmt man den *Business Value* (als keine Struktureigenschaft) einmal außen vor, stehen *Agility* und *Resilience* für die Anpassbarkeit und Widerstandskraft eines Systems, die beiden Werte, welche nicht nur bei Softwaresystemen schon seit Ewigkeiten bei einer Evolution entscheidend sind. Dabei spiegeln diese Werte wieder, inwieweit das System auf Änderungen reagieren kann und resistent gegen schädliche Einflüsse ist. Jedoch reicht es normalerweise nicht, diese Werte einmal anfangs etabliert zu haben. Es muss ständiger Aufwand hinein gesteckt werden, damit beide Werte wenigstens gleichbleibend erhalten werden, da sonst ein Verfall im Sinne der *Architecture Erosion* und *Technical Debt* droht. Im Kontext der Softwareentwicklung kann das dadurch definierte System ebenfalls nicht alleinstehend ausgebildet werden, sondern verlangt nach einer klaren Reihenfolge des Entwicklungsprozesses, um *Agility* und *Resilience* auch zu handhaben. Dies ist in Abbildung 2.6 veranschaulicht, wie es in Furrer [19] beschrieben wurde. Hierbei ist ersichtlich, dass ein zukunftsfähiges System direkt von der Struktur (*langlebige Software*) und den Qualitätseigenschaften (*Agiliy*, *Resilience* und weitere) abhängig ist. Diese werden wieder selbst durch andere Faktoren bestimmt, welche also alle indirekt Auswirkungen auf die Zukunftsfähigkeit besitzen. Der Entwicklungsprozess selbst ist dabei zwar ein zentraler Punkt, aber keineswegs der einzige wichtige Einflussfaktor. Wie man ebenfalls erkennen kann, sind die Treiber des zukunftsfähigen Systems zweierlei Dinge: erstens der *Business Case* definiert die Anforderungen und damit den Rahmen der Software, und zweitens der *Software Engineer*, welcher das System entsprechend konstruiert.

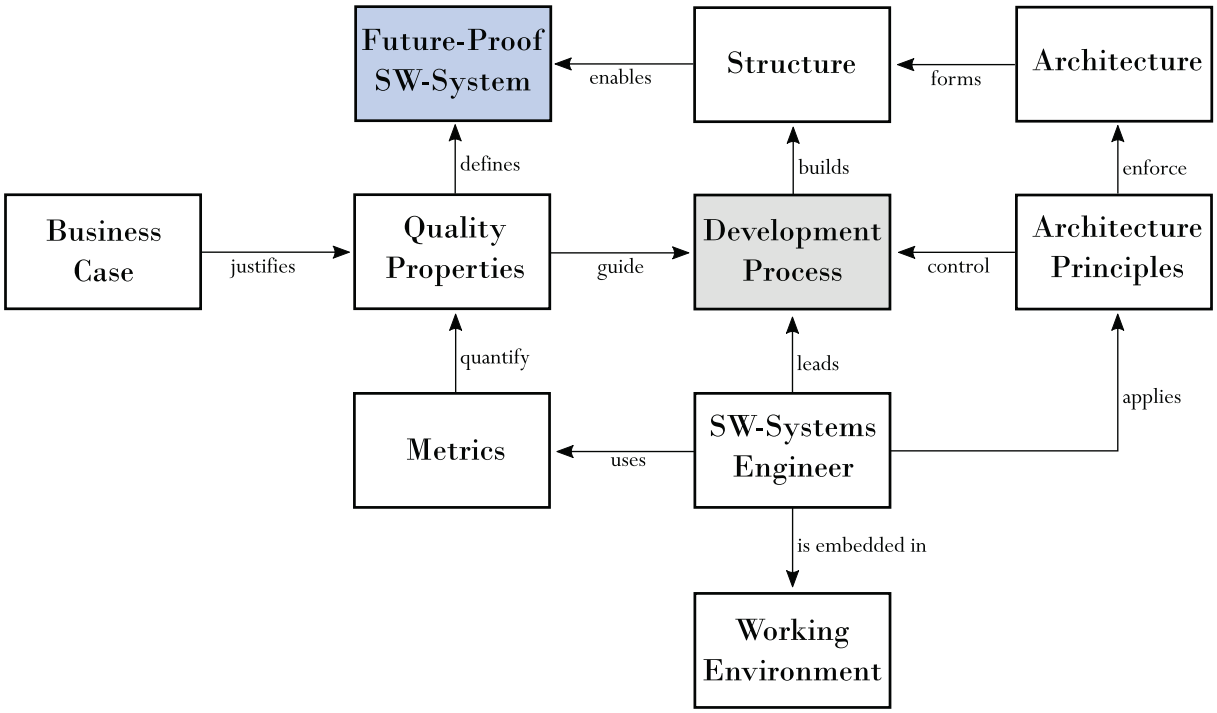


Abbildung 2.6: Die Zusammenhänge bei der Entwicklung neuer, zukunftsfähiger Software

2.4.3 Managed Evolution

Das Vorgehen, die Richtung der Entwicklung sowie die Evolution eines Systems in eine zukunftsfähige Bahn zu lenken, ist derzeit noch ein Gebiet von dünn besetzten Forschungsergebnissen. Es gibt derzeit etablierte Projektmanagement-Prozesse, welche die agile Entwicklung von Software ermöglichen, wie zum Beispiel *Scrum* oder *XP*, jedoch zielen diese nicht notwendigerweise auf den Erhalt von *Agility* und *Resilience* bei einer gleichzeitigen Erhöhung des *Business Value* ab, sondern eher auf eine möglichst kurze *TtM* bei hoher Anforderungsdynamik. Zudem hören diese Management-Typen dort auf, wo die eigentliche Evolution erst beginnt.

Derzeit gibt es drei etablierte Management-Typen (beziehungsweise Vorgehensweisen), welche die Evolution eines Systems handhaben [41]. Der erste Ansatz ist der sogenannte *Greenfield Approach*, welcher im Kern bedeutet, dass alte Teile des Systems einfach durch neue ersetzt werden. Sollten im Lauf der Zeit diverse Teile des Systems der *Technical Debt* oder *Architecture Erosion* unterliegen, wird eine Neuentwicklung dieses Teiles vom System in Betracht gezogen und wirtschaftlich abgewägt. Der Vorteil dieser Methode ist, dass sowohl keine *Adaptive Maintenance* als auch keine *Preventive Maintenance* betrieben werden muss, wie sie in Kapitel 2.1.2 „Die Kosten eines Systems“ vorgestellt wurden. Das bedeutet, dass neue Anforderungen nicht in das System eingepflegt werden müssen und auch keine proaktive Korrektur hinsichtlich Architektur und Design des Systems eingeplant wird. Was damit auf den ersten Blick vielleicht attraktiv wirkt, ist bei genauerer Betrachtung jedoch für die meisten Fälle von großen Systemen eher ungeeignet. Das ergibt sich einfach aus zwei Fakten:

1. das System besitzt hohe Kosten wegen der ständigen Neuentwicklung ausgedienter Teile des Systems, und

2. das System benötigt höheren Aufwand und Investitionen, da zeitweise doppelter Aufwand betrieben werden muss, um die alten Teile des Systems zu betreiben, während die neuen parallel entwickelt werden.

Damit ist diese Methode weitestgehend unwirtschaftlich für große Systeme, da diese nicht einfach so ausgetauscht werden können und die beiden genannten Fakten das System unwirtschaftlich machen würden. Dafür ist diese Methode für kleinere Projekte geeignet, wie zum Beispiel der Entwicklung von Apps für das Smartphone, bei dem eine Neuentwicklung meist wirtschaftlich lukrativer als ein komplettes Rearchitecting ist. Ein anderes Anwendungsgebiet ist in Projekten, in denen Outsourcing betrieben wird oder andere Komponenten von Dritten im eigenen System benutzt werden, dafür jedoch bewusst die Kontrolle des eigenen Systems abgegeben wird.

Der zweite etablierte Ansatz des Managements von Evolution ist der *Microservice Approach*. Hierbei wird das bestehende System in so kleine und unabhängige Teile „zerlegt“, bis diese autark für sich selbst agieren können und spezielle Services nach außen hin anbieten, anstatt eine riesige Monolith-Anwendung zu nutzen. Anschließend werden für diese einzelnen und losen Service-Komponenten separate Vorgehensweisen hinsichtlich der *Adaptive*, *Corrective* und *Preventive Maintenance* betrieben. Das bringt Vorteile in der Wartung und Pflege, in der Implementierung neuer Anforderungen und dem Management von *Technical Debt* und *Architecture Erosion*. Damit eignet sich diese Methode besonders für Systeme, die verschiedene Aufgaben erledigen und in denen das Aufteilen der Komponenten funktioniert, zum Beispiel bei der Kettenverarbeitung von Daten. Das Problem hierbei ist, dass dies eine ungeeignete Methode für Systeme darstellt, welche eine hohe Abhängigkeit zwischen den Komponenten (*Coupling*) im System aufweisen und starken Zusammenhalt (*Cohesion*) dieser benötigen. Diese können nicht einfach „zerlegt“ werden, und selbst wenn es möglich wäre, würden eventuell Qualitätsattribute wie Performance und Usability darunter leiden, was sich bei großen Softwaresystemen schlecht auf die Unternehmen auswirken kann, bei denen es um genau diese Attribute geht.

Ein dritter und neuerer Ansatz ist das auf Evolution konzentrierte Management, die *Managed Evolution*. Die erste Nennung dieses Begriffs wurde von Murer *et al.* [41] aufgenommen und spezifiziert. Die Idee hinter dem Begriff ist, dass bei der Evolution eines Systems nicht immer auf eine teilweise oder komplette Ersetzung dessen gehofft werden kann, da die wirtschaftlichen Folgen zu extrem ausfallen würden. In den meisten Fällen sind kontinuierliche Veränderungen am System besser geeignet. Dabei zielt dieses Konzept darauf ab, stetig neuen *Business Value* zu liefern, während gleichzeitig die *Agility* intakt gehalten und erhöht oder mindestens auf gleichem Level gehalten wird. Das Ziel ist dabei, durch die stetig angepassten Werte für *Agility* mehrere große Restrukturierungs-Programme einzusparen. Dafür wird ein entsprechender *Evolution Channel* aus den Dimensionen *Business Value* und *Agility* (respektive *Resilience*) gebildet. Dabei ist der Management-Part derjenige, welcher dafür sorgt, dass der Verlauf innerhalb der oberen und unteren Grenze des *Evolution Channel* bleibt (vgl. Abbildung 2.7).

Das zugrundeliegende und generelle Vorgehen der *Managed Evolution* ist dabei in Murer *et al.* [41] wie folgt auf die drei Punkte aufgeteilt:

1. Das Auswählen von adäquaten, kleineren *Evolution Steps*. Dabei sollte die Größe und Anzahl dieser Schritte ein überschaubares Maß annehmen.

2. Das *Investment* möglichst balanciert auf die Bereiche der neuen Funktionalität, neuen Anforderungen, steigender *Agility*, Technik-Modernisierung, Reengineering und Architektur-Verbesserung aufteilen.

3. Die Entwicklung geeigneter Metriken für die Überprüfung der oben genannten Bereiche, welche später während der *Managed Evolution* kontrolliert werden.

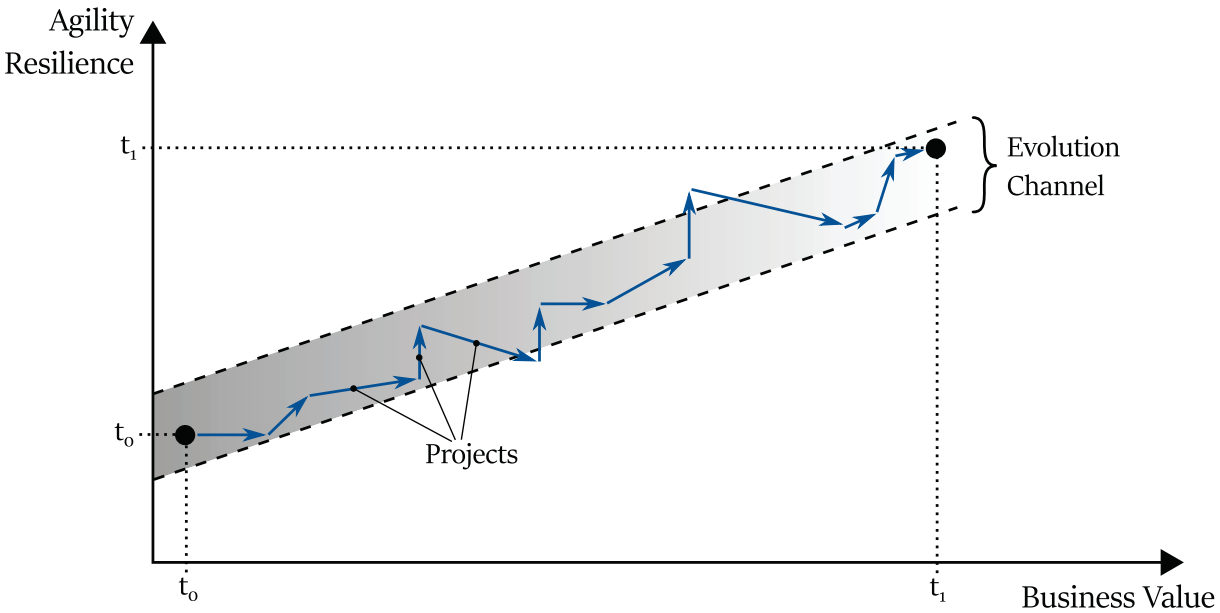


Abbildung 2.7: Die Kernidee der *Managed Evolution*: der *Evolution Channel*

Durch die flexible und systematische Vorgehensweise in der *Managed Evolution* ist ein hoher Grad an Kontrolle der Evolution eines Systems möglich. Dies bezieht die bereits bekannten Architekturprinzipien ein, welche für jeden der *Evolution Steps* beachtet und umgesetzt werden müssen. Damit werden die Prinzipien durch ständige Repräsentation als Richtlinien-Katalog angesehen, welche sich während der gesamten *Managed Evolution* entfalten und die *Agility* sowie *Resilience* des Systems erhöhen. Damit gilt die *Managed Evolution* als Gesamtstrategie für die Evolution des Systems, da sie das Management und die Kontrolle/Messungen der Evolution in Systemen regelt.

Die hier angesprochene *Managed Evolution* hilft dabei, das ausgewählte Prinzip der Redundanz-Freiheit während der Entwicklung und Evolution zu gewährleisten. Jedoch besitzt auch dieses Modell die gleichen Probleme wie andere Vorgehensweisen derzeit: Es fehlt an einem sicheren Nachweis, dass die Arbeit und der Mehraufwand dazu gerechtfertigt sind. Gerade die Investitionen für die Art von Projekten, die im Verlauf in Abbildung 2.7 „nur“ senkrecht nach oben weisen (das heißt, welche *Agility* oder *Resilience* steigern können, jedoch keinen *Business Value* generieren) benötigen erhöhte Überzeugungskraft gegenüber Personen, welche das System aus anderen Blickwinkeln sehen, wie zum Beispiel gegenüber den Programmierern oder dem Business Management. Mit solch einem Nachweis, wie er bereits in Kapitel 1.3 „Zielsetzung“ beschrieben wurde, wäre eine glaubwürdigere Überzeugungskraft und besseres Management möglich. Kombiniert mit dem Vorgehen der *Managed Evolution* wäre damit eine bewährte Basis für Softwareentwicklung und -Evolution gegeben.

3 Methodik und Metriken

Einen noch unbekannten Nachweis über eine bereits vermutete Eigenschaft zu führen, ist bekanntermaßen keine einfache Aufgabe. Dieses Kapitel soll dazu dienen, den Verlauf der Überlegungen und Vorgänge genauer zu beschreiben und aufzuzeigen, inwieweit man Design in Software nachweisen und quantifizieren kann. Dabei spielen eine Reihe von Vorüberlegungen eine Rolle, welche in dieser Arbeit nicht mehr aufgenommen wurden, da sie entweder nicht benötigt, anderweitig umgeformt oder weiter vertieft wurden. Daher soll dieses Kapitel einen durchgängigen Weg beschreiben, der zu dem Kernpunkt dieser Arbeit hinleitet: der eigentliche Nachweis, inwieweit sich Redundanz-Freiheit in Systemen kausal nachweisen lässt.

Wie bereits in Kapitel 1.2 „Motivation für den Nachweis von Werten der Architektur“ geschrieben, ist das eigentliche Ziel nicht, einen exakten mathematisch Beweis zu führen, wenn dies überhaupt bei einem derart abstrakten Thema wie diesem möglich wäre. Es steht der rein kausale Zusammenhang mit dessen Quantität im Mittelpunkt, wird dabei aber durchaus der Einfachheit halber in einigen Fällen im Zusammenhang mit den Begriffen „Beweis“ oder „Nachweis“ genannt und benannt. Diese Einschränkung der beiden Begriffe spielt auch in den darauffolgenden Kapiteln weiterhin eine Rolle, wird daher weiterhin nicht extra erwähnt.

3.1 Methodik-Entwicklung

Der erste Schritt in der Entwicklung einer kausal belastbaren Kette von Schlussfolgerungen ist, eine Methodik zu finden oder zu erstellen, welche geeignet wäre, diesen Nachweis zu führen. Dabei kann die Art, auf welche man den Nachweis führt, sehr unterschiedliche Auswirkung auf den Ausgang und damit der Aussagekraft des Nachweises haben.

Dabei sind zwei wichtige Themen zu klären, welche betrachtet werden müssen, bevor ein noch unbekannter Nachweis erfolgen kann: die Fragen nach der *Darstellbarkeit* und nach der *Messbarkeit*. Beides kann verschiedenste Ausmaße annehmen und unterschiedlichen Ansprüchen unterliegen.

Dabei ist die *Darstellbarkeit* des Problems als erster Punkt dafür wichtig, das eigentliche Problem und eventuelle Lösungen dafür abbilden zu können. Dabei kann es unterschiedliche Ansätze geben, wie man das eigentliche Problem wiedergeben möchte. Dafür sei hier das einfache Beispiel eines mathematischen Beweises, dem „Satz von Euklid“ genannt [65]. Dieser beweist, dass die Anzahl an Primzahlen unendlich ist. Dabei ist die Darstellbarkeit des Problems (die Unendlichkeit der Primzahlen) auf eine Art gelöst, welche einen Raum definiert, in dem die Primzahlen auf einem Zahlenstrahl abgetragen wären. Dies ist eine rein geistige Vorstellung, dennoch ist dieser konkrete Darstellungsraum ein valides Konzept zur Darstellung des Problems mit der Unendlichkeit. Dass dieses Konzept funktioniert, liefert der Beweis selbst, denn zusammen mit der Methode des „Widerspruchsbeweis“ [57] kann man mit dieser Darstellung des Problems den eigentlichen Nachweis hinführen. Im Gegensatz dazu wäre bei einem „Beweis für die Innenwinkelsumme eines Dreiecks mit 180°“ die Darstellung des Problems [51] eine einfache geometrische Konstruktion, auf der man den Nachweis führen kann.

Die *Messbarkeit* beschreibt anschließend an die Darstellung den Nachweis über verschiedene

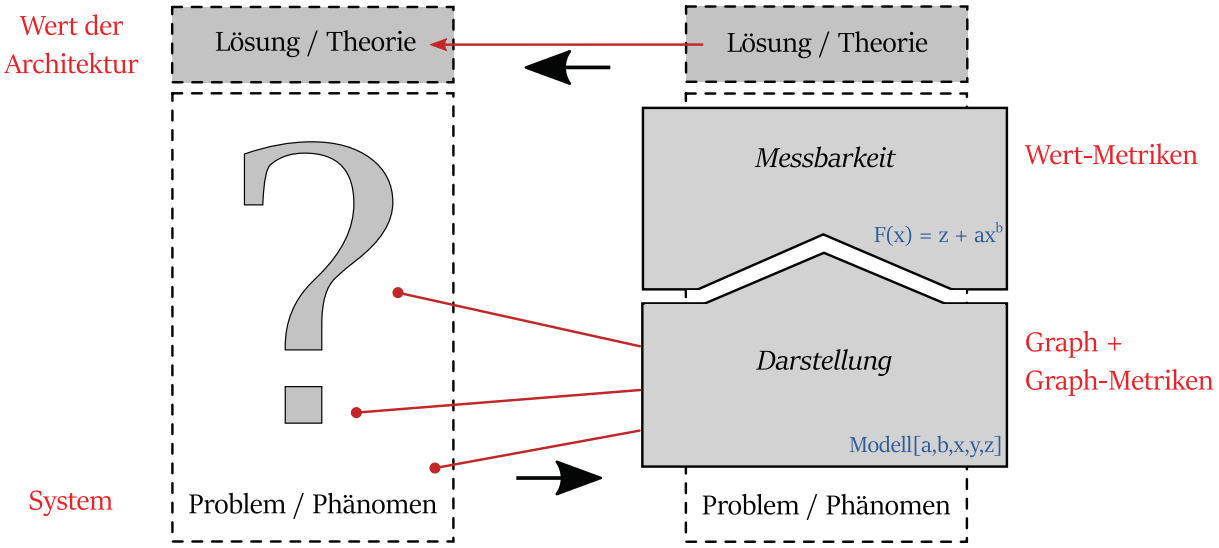


Abbildung 3.1: Das Prinzip *Darstellung und Messbarkeit* vom Problemraum zur Lösung

kausal belastbare Schritte bis hin zum gewünschten Ergebnis. Dabei ist Messbarkeit als Begriff eventuell etwas irreführend, da nicht zwingend ein Messen erforderlich ist, wie zum Beispiel bei dem oben angesprochenen „Satz von Euklid“, welcher auf reine Schlussfolgerung aus dem Darstellungsraum und andere, mathematisch bewiesene Formeln und Axiome setzt. Jedoch wird gegensätzlich dazu im Beweis um die Innenwinkelsumme eines Dreiecks durchaus mit Maß an dem Winkel gearbeitet, auch wenn, wie es in der Natur eines Beweises liegt, keine konkreten Winkel sondern variable Werte für die Maße zu nutzen sind, um deren Allgemeingültigkeit zu garantieren. Dabei ist das Messen an sich nicht immer für den eigentlichen Nachweis relevant, sondern nur für eine Quantifizierung innerhalb des Beweises, wie bei der konkreten Höhe der Winkelsumme. Bei dem Primzahl-Problem existiert damit keine Veranlassung, irgendetwas (außer der Unendlichkeit) aus dem Beweisresultat zu quantifizieren.

In Abbildung 3.1 ist dargestellt, wie dieses Konzept von *Darstellung und Messbarkeit* funktionieren soll. Zuerst steht das Problem (links unten), beziehungsweise das beobachtete Phänomen im Raum. Um die zugehörige Lösung oder Theorie zu entwickeln, werden aus dem Problemraum die notwendigen und sinnvoll erscheinenden Kernpunkte extrahiert, dargestellt als rote Kanten in der Grafik. Damit ergibt sich die Darstellung, indem das Phänomen als Modell oder im neuen Darstellungsraum abgebildet wird. Schematisch wird dies mit dem blau markierten $Modell[a, b, x, y, z]$ dargestellt, was bedeuten soll, dass eine Darstellung gewählt wurde, welche die Kernpunkte a, b, x, y, z aus dem Phänomen betrachtet. Darauf aufbauend beginnt die Messbarkeit, welche mit den in der Darstellung gewählten Kernpunkten hantiert. Mit diesen Kernpunkten wird die später resultierende Theorie erklärt und hinreichend kausal belegt. Dies ist in der Grafik beispielhaft mit $F(x) = z + ax^b$ dargestellt, was bedeuten soll, dass in der Messbarkeit die Zusammenhänge der Kernpunkte aus der Darstellung beschrieben werden. Ist dies erfolgt, kommt man zu einem Resultat als Ende der Messbarkeit, welches sich zurück auf das ursprüngliche Phänomen und dessen Theorie projizieren lässt. Die dadurch entstandene Lösung ist nun, wenn alle Schritte logisch nachvollziehbar sind, die Bestätigung der anfangs geahnten und nun nachgewiesenen Theorie. Meistens werden in der Praxis noch empirische Daten oder mathematische Beweise innerhalb der Messbarkeit einbezogen, um die Aussagekraft zu stärken.

Die Frage, welche sich im Rahmen dieser Arbeit ergab, ist, inwieweit das Vorgehen mit der

Darstellung und Messbarkeit bei Nachweisen mit Design von Software noch handhabbar bleibt. Aber auch hier hat sich ergeben, dass das Konzept von Darstellen und Messen noch anwendbar ist, wenn auch nicht in dem mathematischen Sinne wie bei den beiden oben beschriebenen Beweisen. Denn auch ein Problem von Architektur und Design kann man auf einen Darstellungsraum abbilden und, wenn geeignet gewählt, daran die entsprechenden Messungen und kausalen Schlussfolgerungen ziehen, wie es die rot markierten Wörter in Abbildung 3.1 angeben. Dabei sollen die beiden folgenden Abschnitte genauer erläutern, wie und in welchem Ausmaß die Darstellbarkeit und Messbarkeit in Bezug auf Softwarearchitektur und -design angewendet werden kann.

3.1.1 Darstellbarkeit

Die Darstellbarkeit von Software ist bereits vielerorts näher beleuchtet, zum Beispiel durch UML (*Unified Modeling Language*), ADL (*Architecture Description Language*) oder Ontologien. Es bleibt jedoch die Frage offen, inwieweit die besagten Sprachen und Beschreibungen für einen kausalen Nachweis tauglich sind, da zwar alle Methoden ihrem eigenen Zweck dienlich sind, jedoch nicht für Beweislasten ausgelegt sind. Abgesehen davon bieten die angesprochenen Softwarebeschreibungen bereits viele Elemente, welche die Darstellungen von Design-Problemen zumindest teilweise ermöglichen. Als Beispiel sei hier die verbreitete Beschreibungssprache UML genutzt. Sie bietet Möglichkeiten, Probleme von Struktur innerhalb einer Software in vielen Fällen konkret darstellen zu können. Doch auch UML hat Probleme bei genauer Beschreibung abseits der funktionalen Ebene, zum Beispiel bei zeitlichen Aspekten, der Definition von Qualitätsattributen oder höheren Ebenen des Softwaredesigns, abseits von Klassen und Methoden. So wie UML hat fast jede auf dem Markt etablierte Sprache oder Methode ihre Grenzen, was jedoch in vielen Fällen kein Problem darstellt, sind sie dennoch für die Zwecke geeignet, für die sie geschaffen wurden. Der eigentliche Punkt, welcher für einen kausalen Nachweis von Softwaredesign erforderlich ist, ist die geeignete Abbildung des Darstellungsraumes in ausgewählte Teile von einer oder mehreren existierenden Methoden. Sollte sich dies als ungeeignet herausstellen, kann eine Kombination oder Neuentwicklung in Betracht kommen, um eine für den Nachweis geeignete Darstellung des Problems anbieten zu können, was wiederum als Grundlage für die spätere Messbarkeit dient.

Die Reichweite der Möglichkeiten, eine Software strukturiert darzustellen und als mögliche Grundlage für kausale Zusammenhänge zu nutzen, ist reichlich (UML, Petri-Netze, ADL, Meta-Modelle, Ontologien, (Component-)Maps, Flowcharts, strukturierte Bäume und Listen, und weiteres). Dabei gilt es bei der Auswahl einer Methode der Darstellung immer, das eigentliche Ziel des Nachweises zu bedenken, denn im Endeffekt muss mittels der Darstellung das Ziel später sinnvoll nachgewiesen werden. Die Auswahl und Eingrenzung auf eine bestimmte Darstellung ist ein wichtiger, aber insbesondere auch schwieriger und kreativer Part bei einer Beweisführung. Der Darstellungsraum legt fest, in welche Richtung der Nachweis geschieht, zum Beispiel über welche anderen Definitionen man seine Theorie bestätigt oder welche Konstruktion geeignet wäre, die Lösung kausal und unwidersprochen darzulegen.

Eine Analyse der Thematik dieser Arbeit ergab, dass eine Neuentwicklung einer Darstellung auf Grundlage bestehender Methoden das geeignete Mittel sein könnte, um *Architecture Principles* in einem System darzustellen. Die bestehenden Methoden und Techniken, besonders die des UML, beeinflussten die Entscheidung, das System als eine Form eines strukturierten Graphen darzustellen. Dabei wurden Anpassungen vorgenommen, welche über den Zweck und der Machbarkeit mittels UML hinausgingen, was die erwähnte Neuentwicklung nach sich zog. Gleichzeitig wurden viele Elemente etablierter Graph-Modelle entfernt, um nur unbedingt not-

wendige Elemente zu nutzen. Die Entwicklung und der entstandene Graph werden dabei in Kapitel 3.2 „Softwaresysteme als Graphen“ genauer erläutert und konkreter beschrieben. Mittels dieses entwickelten strukturierten Graphen ist es möglich, das Ziel, den Wert von Software durch Architektur nachzuweisen, kausal und anschaulich zu erreichen. Jedoch ist der neue Graph bislang nur die Darstellung des Nachweises, die Messbarkeit soll der folgende Abschnitt näher beleuchten.

3.1.2 Messbarkeit

Ist die Darstellung einer Beweisführung erst einmal gewählt, ist die nächste Frage die nach der Messbarkeit innerhalb der Darstellung. Dabei ist es sinnvoll, den Fakt, dass eine möglichst sinnvolle Messbarkeit erreicht werden muss, bereits bei der Wahl der Darstellung mit einzubeziehen. Genau wie es bei der Entscheidung einer geeigneten Darstellung mehrere Varianten geben kann, so kann auch die Methode der Messbarkeit sich untereinander unterscheiden. Es gibt sicherlich in der oben genannten Beweisführung zu der Innenwinkelsumme eines Dreiecks mehrere Möglichkeiten, in dem dort gewählten geometrischen Darstellungsraum die Innenwinkelsumme geometrisch eindeutig nachzuweisen.

Die Frage nach der geeigneten Methode für Messungen in einer Darstellung ist am Ende eine, welche von sinnvoller Umsetzung sowie persönlichem Geschmack abhängt. Während man bei der Wahl der Darstellung das gesamte Problem erfassen muss und das Problem selbst nicht anpassen kann, ist es zumindest geringfügig möglich, die bereits gewählte Darstellung bezüglich der Messbarkeit anzupassen. Wie bereits weiter oben beschrieben, ist nicht immer davon auszugehen, dass Messbarkeit eine reine Messung innerhalb des erstellten Modells ist. Viel eher ist die Messbarkeit ein Prozess, in welchem man die Eigenschaften des gewählten Modells (beziehungsweise der Darstellung) ausnutzt, um sein Ziel zu erreichen.

Die Messbarkeit in Software ist ein eigens geschaffenes Problem, welches es mittels Nachweis zu lösen und validieren gilt. Der Vorteil von Messbarkeit in Software ist, dass man hierbei keine eventuell vorhandenen unbekannten Phänomene wie in der Physik antrifft, sondern sich alles über die Art des Designs und der Programmierung belegt. Der Nachteil ist dabei jedoch, dass der Nachweis auch weniger exakt sein wird, da hierbei nur schwerlich auf Naturgesetze für die Messbarkeit zurückgegriffen werden kann. Das bedeutet nicht, dass eine Messbarkeit nicht möglich ist, jedoch wird sich der Ausgang der Messbarkeit und damit das Resultat des Nachweises auf eine viel erheblichere Weise positiv oder negativ ändern, je nachdem, welche Methode der Messbarkeit eine Anwendung findet. Das bedeutet durchaus, dass es einfacher ist, aus einer gleichen Darstellung und verschiedenen Messbarkeiten auch unterschiedliche Ergebnisse aus dem Problem zu ziehen. Deswegen ist bei derart nicht-mathematischen oder ähnlichen Beweisen darauf zu achten, mit der richtigen Begründung Darstellungen und Messbarkeiten zu wählen.

Die Messbarkeit in Software beinhaltet zwar verschiedene Methodiken, jedoch ist es in der Softwaretechnik so gut wie unmöglich, die Messbarkeit anders als ein echtes Messen zu verstehen. Gemessen wird dann innerhalb der Darstellung, beziehungsweise verschiedenste *Metriken* innerhalb der Software. Dafür gibt es bereits eine Menge an praktischer Literatur [16] [17] [35], welche gute Ansätze für geeignete und etablierte Metriken anbietet. Dabei gibt es Metriken für alle möglichen Ebenen und Abstraktionen der Software. Die bekannteste Metrik, welche in obigen Kapiteln bereits einige Male erwähnt wurde, ist die Anzahl an Quellcodezeilen *#LOC* als ein Maß für die Größe von Software. Man kann erahnen, dass diese Metrik nur bedingt für alle Softwaretypen und Nachweise geeignet ist, da alleine die Definition, was unter einer Zeile an Quellcode zu verstehen ist, verschiedene Ansichten anbietet. Es ist jedoch das Ziel des Nachweises und eine Aufgabe der Wahl einer geeigneten Messbarkeit in der Darstellung, solche Definitionen

als Grundlage für den Nachweis festzuhalten, derart, dass das Ergebnis des Nachweises nicht durch eine falsche Annahme einer Metrik ausgehebelt wird, sondern die kausale Kette erhalten bleibt. Sollte für ein Problem keine bereits existierende Metrik vorhanden sein, ist es möglich, eine eigene Metrik (durchaus auch auf anderen Metriken aufgebaut) zu definieren. Aber auch dabei ist auf die Standhaftigkeit der neuen Metrik innerhalb der Beweiskette zu achten. Ein Beispiel wäre hier die Metrik der *Test Coverage*, ein Maß für die prozentuale Abdeckung von Funktionen mit Tests. Dies wird allgemein als Metrik für Softwarequalität benutzt, ist auf der anderen Seite jedoch für sich allein stehend nur ungeeignet, die *Technical Debt* in einem System zu quantifizieren. Jedoch wäre es durchaus möglich, mit guter Argumentation eine neue Metrik zu definieren, welche die *Test Coverage* als Grundlage mit einbezieht.

Für das bearbeitete Problem dieser Arbeit wurden beide Fälle genutzt, sowohl das Nutzen bereits bekannter, als auch das Entwickeln neuer Metriken. Dabei wurde immer darauf geachtet, dass beides im Zusammenhang mit dem Problem steht und sich durch die gewählte Darstellung als strukturierten Graph abbilden lässt.

3.2 Softwaresysteme als Graphen

Wie in dem vorherigen Kapitel beschrieben, wird in dieser Arbeit das Softwaresystem als strukturierter Graph benutzt, um eine geeignete Darstellung des Nachweises zu erhalten. Dies ist jedoch nicht unbedingt die einzige Darstellungsmethode, Systeme können durchaus auch durch andere Methoden dargestellt werden, dabei seien hier exemplarisch Petri-Netze, UML-Diagramme (zum Beispiel Klassen-, Aktivitäts- oder Sequenz-Diagramm), Architektur-Beschreibungssprachen, formale Modelle und ähnliche Methoden genannt. Jede Möglichkeit ist umfangreich in der Literatur vertreten, weshalb in dieser Arbeit nicht genauer auf diese eingegangen wird. Einen geeigneten und weiterführenden Überblick über die Modellierungen einer Problematik gibt zudem Kastens *et al.* [27], wo darauf eingegangen wird, inwieweit das Modellieren eines Szenarios formal beschrieben werden kann.

Eine geeignete Auswahl einer Darstellungsmethode ist essenziell, um später den Nachweis daran durchzuführen. Bei der Wahl einer geeigneten Darstellung in dieser Arbeit wurden einige Voraussetzungen getroffen, welche ausreichend erfüllt sein mussten. Dabei war es wichtig, dass die Darstellung

a) die relevanten Eigenschaften des Systems realistisch widerspiegelt,

b) verschiedenste Ausprägungen des Systems erfassen kann,

c) hinreichend formal beschreibbar ist, um diese eindeutig zu erhalten und Messbarkeit zu ermöglichen, und

d) die Messbarkeit muss in diesem formalen Modell geeignet und nachvollziehbar umsetzbar sein.

Diese vier Punkte definierten die Menge an zukünftigen Modellen, welche für die Darstellung im Nachweis genutzt werden soll. Die Punkte (a) und (b) sind insofern wichtig, damit die gewählte Darstellung überhaupt einen Sinn ergibt. Ein formales Modell, welches nicht ausreichend das System widerspiegelt, ist zwar messbar, jedoch stimmt damit die Annahme des Nachweises nicht mehr, was den ganzen Nachweis an sich ungültig macht. Welche Eigenschaften und Ausprägungen die Darstellung dann besitzt, ist jedoch eine andere Frage. Dies ist dann abhängig von der gewählten Methode des Nachweises, dem Ziel sowie der Methode der Messbarkeit. Auch

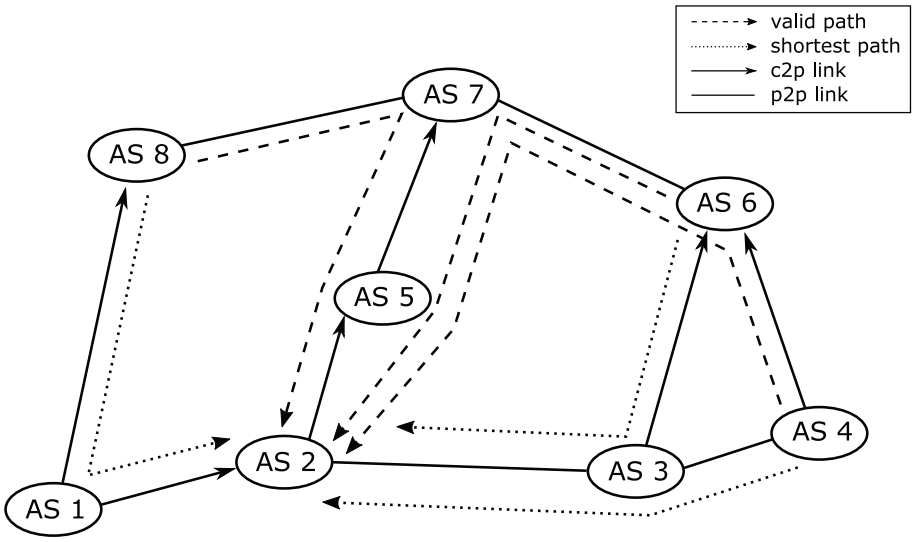


Abbildung 3.2: Ein Beispielgraph für Netzwerke aus Dimitropoulos *et al.* [14]

in Architektur und Design von Software können verschiedene Wege das Ziel des Nachweises erreichen. Aus Punkt (b) folgt fast zwangsweise der dritte Punkt (c), welcher festlegt, dass ein Modell für ein System bezüglich der Kernpunkte des Systems eindeutig sein soll, sowie ebenfalls ein System eindeutig in ein Modell abbildbar sein soll. Nur dies garantiert einen standfesten Nachweis. Der Punkt (d) ist dann eine Voraussetzung für eine sinnvolle Folge aus dem Modell, da später anhand der gewählten Darstellung die Messungen durchgeführt werden müssen.

Einige der oben genannten möglichen Modelle schieden durch die Definition der Punkte (a) bis (d) von vornherein aus. So ist zum Beispiel die Darstellung mit gefärbten Petri-Netzen nicht sinnvoll erschienen, da dies die Punkte (a) und (b) nicht gut erfüllt. Dabei sind Petri-Netze zwar sehr formal beschreibbar, jedoch nur bedingt geeignet für die Darstellung von Architektur und Design von Softwaresystemen, besonders in Hinblick auf die dafür notwendigen Eigenschaften für den Nachweis (Software-Metriken). Ebenso ist ein Klassendiagramm oder Aktivitäten-Diagramm in UML als Darstellung zwar möglich, jedoch nicht hinreichend deckend mit Punkt (c), da UML nicht als eindeutig bezeichnend für ein System wirkt. Es fehlt den UML-Diagrammen an geeigneter Fähigkeit für Abstraktion und Einheitlichkeit für Darstellungen bestimmter Eigenschaften. Eine spätere formale Messbarkeit in UML-Diagrammen würde vermutlich an nicht ausreichend definierten Elementen oder deren Mehrdeutungen scheitern.

Aus diesem Grund wurde im Rahmen dieser Arbeit entschieden, einen strukturierten Graph als Darstellungsform für den Nachweis zu schaffen. Bei der Auswahl des Graphen wurden alle vier oben genannten Punkte berücksichtigt. Das Resultat ist ein *labeled, annotated, directed Multigraph*, auf welchen genauer in Kapitel 5 „Der Nachweis“ eingegangen wird. Dieser Graphentyp vereint alle notwendigen Eigenschaften, die für den Nachweis erforderlich sind und ist gleichzeitig einer Vereinfachung unterlaufen, sodass nur die wirklich relevanten Eigenschaften des Systems abzubilden sind. Softwaresysteme als Graphen darzustellen ist generell keine neue Idee, da auch Petri-Netze und UML im groben Sinn als Graphen klassifiziert werden können. Der Vorteil bei der Nutzung von Graphstrukturen ist dabei, dass gleichzeitig Knoten und Kanten als Elemente mit Beziehungen dargestellt und eigens definiert werden können. Damit können Strukturen des Systems in den meisten Fällen abgedeckt werden, wie zum Beispiel die verschiedenen Möglichkeiten der Interaktion von Elementen im System. Eine solche Beispielstruktur für einen

Graph ist aus Dimitropoulos *et al.* [14] entnommen und einmal in 3.2 abgebildet, auf den später auch noch einmal eingegangen wird. Man kann erkennen, dass die Autoren einen strukturierten Graph wählten, um ihre Version einer Darstellung für Netzwerke aufzuzeigen. Auch wenn deren hier gezeigter Graph nicht der Kern ihrer Arbeit ist, stellt dieser hier jedoch ein gutes Beispiel dar. Graphen im Allgemeinen sind im Grunde immer geeignet für die Punkte (a) und (b), da sie flexibel anpassbar, je nach Struktur des Systems, sind. Der Punkt (c) ergibt sich aus der anschließenden Definition des Graphen, dessen Knoten und Kanten, sowie deren Bedeutungen. Der letzte Punkt (d) ist dann das Resultat aus den in (c) gesetzten Formulierungen, das heißt, inwieweit die festgelegten Größen und Eigenschaften der Elemente im Graph für den eigentlichen Zweck der Messbarkeit von Nutzen sind. Schließlich ist ein komplett definierter und mit Bedeutungen versehener Graph nicht sinnvoll, wenn die festgelegten Informationen den Zweck verfehlen. Aus diesem Grund ist die in den vorherigen Kapiteln angesprochene Voraussicht auf die Messbarkeit sowie deren Rückwirkung auf die Darstellung während des Nachweis-Aufbaus nötig, da durchaus neue Erkenntnisse in der Messbarkeit diverse Änderungen an der grundlegenden Darstellbarkeit erfordern, um geeignet schlussfolgern zu können.

Graphen sind selbstverständlich nicht die Universallösung für das Problem einer geeigneten Darstellung, da zum Beispiel die Eigenschaft, verschiedene Schichten in Systeme einzubauen, nicht oder nur schwer in einen Graph umgesetzt werden kann, der noch weitere Eigenschaften besitzen soll. Die Abstraktionshöhe besitzt so auch in Graphen ein oberes Limit in der Darstellung, was jedoch für den Nachweis dieser Arbeit keine Einschränkung darstellt.

3.2.1 Eigenschaften

Wenn ein Softwaresystem als Graph dargestellt wird, werden dadurch sofort einige Eigenschaften mit impliziert, was sich daraus ergibt, wie ein Graph strukturiert ist. Die wichtigste Eigenschaft bei Darstellungen als Graph ist, dass dieser Knoten und Kanten besitzt, die beiden Hauptelemente eines Graphen. Die Definition dieser ist bei Graphen als Softwaresystem in den meisten Fällen als *Element* und *Beziehung* definiert. Was genau die Mengen der Elemente und Beziehungen beinhalten, das ist die angesprochene individuelle Zuweisung für die spätere Messbarkeit. Unter Element kann man verschiedenste Dinge definieren, welche in einem Softwaresystem vorkommen, zum Beispiel Klassen, Entities, Methoden und Funktionen, Interfaces, Module, Plattformen und weiteres. Die Definition für Kanten im Graph wären in einem solchen System dann zum Beispiel eine Assoziation, eine einseitige Benutzung, ein Daten-Transfer, ein ausgelöstes Event oder auch bedingte Abhängigkeit. Die Bedeutung der Elemente im Graph wären im oben eingeführten Beispiel von Abbildung 3.2 dabei, dass die Knoten diverse „Wegpunkte“ beim Routing darstellen und die Kanten verschiedenste Arten von Transfer-Beziehungen sind. Damit kann sehr gut die Infrastruktur dargestellt werden und eine Messbarkeit wäre darauf aufbauend möglich, wenn der Graph als Darstellung für ein Problem dient.

Der Detailgrad bei einem Graphen, welcher Software modelliert, kann extrem variieren. Die einfachste Form wäre ein Graph, dessen Knoten und Kanten jeweils eine Definition zugewiesen bekommen haben. Dabei gäbe es nur eine einzige Art von Knoten sowie Kanten ohne jegliche Erweiterungen. Anschließend lässt sich der Detailgrad in beiden Fällen, bei Knoten und Kanten, unabhängig steigern, wie es in Abbildung 3.3 schematisch aufgezeigt ist. Eine solche Steigerung bei Kanten wäre zum Beispiel, normale Kanten auch gerichtet zuzulassen. Solche gerichtete Kanten sind bei Graphen keine Seltenheit, da sie sich sehr gut für bedingte einseitige Beziehungen eignen, wie zum Beispiel der Zugriff auf eine Ressource. Eine weitere Möglichkeit, den Detailgrad einer Kante zu steigern wäre, verschiedene Typen von Kanten einzuführen. Damit ergäbe sich ein komplexeres Netzwerk von Knoten und Kanten, in denen die Kanten verschiedene Typen

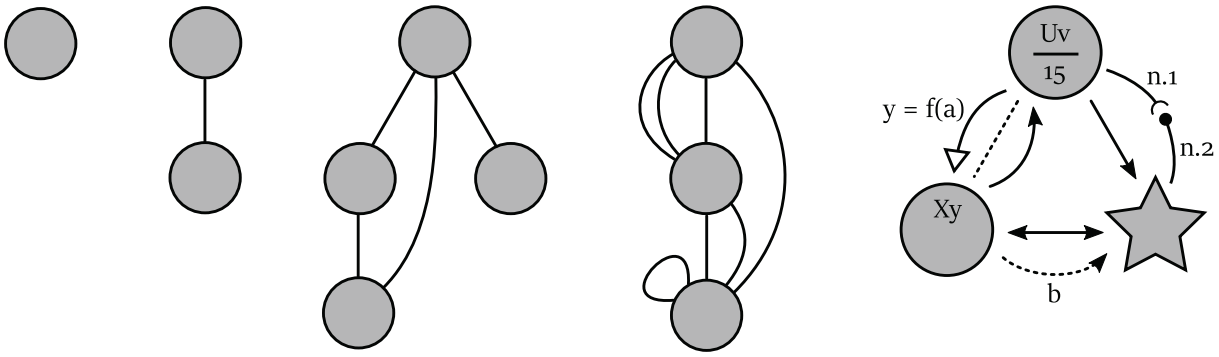


Abbildung 3.3: Verschiedene Möglichkeiten eines Graphen

von Beziehungen darstellen können, zum Beispiel einen Typ für Zugriff, einen für Abhängigkeit. Es sind noch viele weitere Verfeinerungen von Kanten möglich (zum Beispiel parallele Kanten, Kanten mit Eigenschaften, Kanten mit Bedingungen, und weiteres), auf welche hier aber nicht weiter eingegangen werden soll. Man kann in dem Beispiel von Abbildung 3.2 erkennen, dass die Autoren sowohl gerichtete und parallele Kanten einführten, als auch verschiedene Typen von Kanten nutzten. Der Grad der Verfeinerung, welche zur Darstellung des Systems benötigt wird, hängt am Ende von dem Zweck der Verwendung ab. Eine Steigerung des Detailgrades bei Knoten sieht dem der Kanten ähnlich. Auch hier wäre eine typische Verfeinerung, verschiedene Knoten-Typen einzuführen, welche für verschiedene Element-Typen im System stehen. Aber auch die Möglichkeit, dass Knoten weitere Knoten oder ganze Graphen beinhalten können, wäre eine sinnvolle Verfeinerung. Weitergehend können auch Knoten verschiedenste Bedingungen und Eigenschaften aufweisen, um diverse Beschränkungen im System darstellen zu können. In Abbildung 3.2 gibt es nur eine Art von Knoten, da keine weiteren Verfeinerungen zwar möglich, jedoch nicht notwendig waren. Damit hängt auch der Grad der Verfeinerung bei Knoten letztlich von dem Verwendungszweck ab.

Ein Graph ist mit all seinen Eigenschaften in der Mathematik innerhalb der Graphentheorie sehr genau definiert und es würde den Rahmen dieser Arbeit sprengen, hier die komplette Graphentheorie zu betrachten. Wichtig ist, dass ein Graph $G(V, E)$ durch seine Mengen an Knoten V und Kanten E dargestellt wird, darauf arithmetische Mengenoperationen ausgeführt werden können und die entsprechenden Elemente zusätzliche Eigenschaften aufweisen können, wie *Gewichtung*, *Richtung*, *Parallelität* oder *Färbung*, was alles einer Attribut-Bezeichnung der einzelnen Elemente gleichkommt. Da die mathematischen Definitionen nur seltenst in dieser Arbeit benötigt werden, werden die genutzten mathematischen Definitionen an den entsprechenden Stellen in dieser Arbeit noch einmal kurz aufgeführt und beschrieben. Für Interessierte bieten sich folgende Literaturstellen für genauer beschriebene Themen zur Graphentheorie an [31] [55].

Der Detailgrad von den benutzten Graphen in dieser Arbeit ist so einfach wie möglich gehalten, da das dem Zweck eines Nachweises entgegenkommt. Ein Graph mit zu vielen Eigenschaften wäre dahingehend kontraproduktiv für sinnvolle Schlussfolgerungen, da viele Eigenschaften nicht und nur sehr marginal genutzt werden würden. Deswegen wurde sich auf die kleinste Menge an Details beschränkt, bei der der Nachweis noch kausal und quantitativ geführt werden kann. In dieser Arbeit besitzen Knoten verschiedene Typen und jeder Typ besitzt wiederum verschiedene Eigenschaften. Ebenso besitzen Kanten verschiedene Typen, können mehrmals zwischen Knoten auftreten und haben genau wie Knoten auch bestimmte Eigenschaften. Für weitere Informationen hierzu sei noch einmal auf Kapitel 5 „Der Nachweis“ verwiesen.

3.2.2 Berechnungen und Verhalten

Dadurch, dass das System auf einen Graph abgebildet wurde, können durchaus allgemeine Graph-Algorithmen auf das System angewandt werden. Eines der populären Beispiele ist eine Darstellung des World Wide Webs (Internet) als Graph, bei dem dann Algorithmen wie „Kürzeste Wege“, „Minimaler Schnitt“ oder „Maximaler Fluss“ ausgewertet werden können, um entsprechende Eigenschaften in dem System (in diesem Fall im Internet) aufzuzeigen und zu quantifizieren. Bei solchen Algorithmen werden konkret Eigenschaften von Knoten und Kanten ausgenutzt, so kann zum Beispiel der „Maximale Fluss“ nur dann erfolgreich berechnet werden, wenn Kanten auch gerichtet sind und eine Kapazitätseigenschaft besitzen. Hierfür ist vor allem die Definition der Kanten essenziell, der Detailgrad der Knoten hält sich bei diesen Beispielen dabei in Grenzen. Es existiert bereits reichlich Literatur, was Graph-Algorithmen anbelangt [13] [31], weswegen hier weniger darauf eingegangen wird.

Generell gesehen gilt bei Berechnungen und der Auswertung des Verhaltens eines Systems durch Graphen, dass jeweils die nötigen Eigenschaften des Graphen genutzt werden. Dabei spielt wieder der Detailgrad des Graphen eine große Rolle, da dessen Beziehungen zu den Elementen und deren Eigenschaften und Bedingungen ausgenutzt werden. Bei der Darstellung von Softwaresystemen ist es abhängig von der Fragestellung, inwieweit eine Auswertung der Details im Graph eine Rolle spielt. Es müssen nicht zwangsweise mathematische Graph-Algorithmen angewendet werden, meist sind eher einfache Zähloperationen das eigentliche Ziel bei Software-Graphen, wie zum Beispiel die Anzahl der Abhängigkeiten eines Elementes im System. Das wiederum würde bedeuten, dass im Gegensatz zu den mathematischen Algorithmen die Eigenschaften der Knoten viel stärker eingebunden werden würden, die Gestaltung der Kanten jedoch eher vernachlässigbar wäre. Es kommt daher immer auf den Zweck an, wie man diese ausgestaltet. Es ist ebenso ein nicht zu vernachlässigender Faktor, inwieweit sich der Graph dynamisch entwickelt. Die Dynamik eines Graphen reicht von einer kompletten statischen Auswertung bis hin zu Graphen, welche nach bestimmten Mustern sich hoch frequentiert abändern, weil zum Beispiel ein Event getriggert wurde. Bei der Auswertung eines Graphen ist demzufolge genauestens darauf zu achten, welche Möglichkeiten sich durch den nächsten Schritt im Algorithmus ergeben und welche man davon nutzt, sollte dies zur Auswahl stehen.

Im Rahmen dieser Arbeit wird kein komplexer Graph-Algorithmus ausgeführt, sondern lediglich diverse Zähloperationen auf Knoten sowie auf Kanten durchgeführt. Dabei werden die unterschiedlichsten Eigenschaften wiederverwendet, welche auf dem strukturierten Graphen gebildet wurden. Das liegt nicht daran, dass solche komplexen Graph-Algorithmen nicht möglich wären, sondern viel mehr daran, dass für das Ziel des Nachweises kein „Kürzeste Wege“-Algorithmus oder Ähnliches vonnöten ist. Die Dynamik in dem später benutzten Graphen ist insofern nur partiell gestaltet, dass eine Veränderung des Graphen nicht durch den Algorithmus selbst sondern zwischendurch erweitert werden kann und wird.

3.3 Graphen als Werkzeug für Architecture Principles

Ein Thema, welches ebenfalls in dieser Arbeit angesprochen wird, ist das Darstellen von Architekturprinzipien mittels Graphen. Konkret soll das Prinzip der Redundanz im System als Eigenschaft des Systems als strukturierter Graph dargestellt werden, dazu später in Kapitel 4 „Redundanz“ mehr. Um das Konzept eines Graphen zu bedienen, kann man Architekturprinzipien als die Eigenschaft des Graphen verstehen. Ein Graph selbst besitzt nur wenig Nutzen, wenn mit ihm nichts dargestellt oder ausgewertet werden soll, jedoch ist der Sinn durch

die Architekturprinzipien-Darstellung gegeben. Wie genau diese ausschaut, ist Sache des verfolgten Zwecks, aber in jedem Falle im Rahmen des Möglichen. Um ein Architekturprinzip als Graph darzustellen, ist die erste Frage die nach den Graph-Eigenschaften. Die Basis an Eigenschaften, Berechnungen und Verhalten eines Graphen, wie im vorherigen Abschnitt genannt, bleibt dabei bestehen. Für die Darstellung eines Prinzips muss geklärt werden, welche Eigenschaften das Prinzip im System berührt und welche Auswirkungen sich durch das Prinzip im System ergeben. Dabei helfen zum Beispiel eine empirisch ermittelte Differenz-Analyse des Systems oder auch Erfahrungswerte aus vorherigen Systemen. Die so ermittelten relevanten Eigenschaften des Systems müssen anschließend sinnvoll in den Graph eingearbeitet werden. Das bedeutet insbesondere die Definition der Knoten, Kanten und Eigenschaften des Graphen bezüglich dessen, wie die relevanten Eigenschaften des Systems auf den Graph abgebildet werden sollen. Es ist schwer, hier ein Beispiel für eine fehlerhafte Abbildung von Systemeigenschaften auf Graph-Elementen zu finden, da mit einer richtigen Begründung und Intention die gewählte Abbildung wieder vollkommen zutreffend sein kann. Man kann ein existierendes Interface auf einen Knoten abbilden, aber ebenfalls auf eine Eigenschaft eines Knotens, eine Kante oder eine Eigenschaft einer Kante. Nichts davon ist wirklich falsch, es kommt dabei immer auf den Kontext an.

Abgesehen davon, liegt es in der Natur von Softwareprinzipien, dass diese eher abstrakt gehalten sind. Oft steht noch lange vor der Frage nach der Abbildung die Frage nach den Eigenschaften. Wie oben erwähnt, müssen relevante Eigenschaften ausgemacht werden. Das mag bei einigen Prinzipien einfacher sein als bei anderen, jedoch wird es auch für ein Prinzip verschiedene Graph-Ausprägungen geben können, da hierbei viel Spielraum gegeben ist, das Prinzip auf die relevanten Eigenschaften und den dazugehörigen Graphen anzuwenden. Es ist den meisten Architekturprinzipien gemein, dass sie sich auf bestimmte Elemente oder Komponenten im System beziehen und gleichzeitig deren Eigenschaften sowie Beziehungen untereinander gewisse Grenzen setzen. Diese Elemente können in vielen Fällen als Knoten dargestellt werden und die zu begrenzenden Eigenschaften als Kanten. Dies ist zwar eine sehr verallgemeinerte Vorgehensweise, jedoch ist diese bei genauerer Betrachtung durchaus oft auf Architekturprinzipien anwendbar. Ein Beispiel wäre statt der Redundanz-Freiheit das Prinzip der *Re-use and Parametrization*, welches die Wiederverwendung von Komponenten steigern soll. Diese Komponenten wären dann Knoten im Graphen und die Kanten stellen anschließend die Eigenschaft des *Re-use* dar, inklusive der Begrenzung, inwieweit bestimmte Eigenschaften der Knoten (Komponenten) ihr *Re-use* benutzen. Es wären natürlich weitere Definitionen des Graphen erforderlich, doch dieses Szenario wäre ein möglicher Ansatz auf Basis eines strukturierten Graphen.

3.4 Metriken in Softwaresystemen

Die Metriken in einem Softwaresystem stellen in dieser Arbeit die Messbarkeit dar. Metriken sind, wie in vorherigen Kapiteln bereits einige Male erwähnt, Software-Messwerte, welche direkt oder indirekt aus dem System extrahiert werden können. Die Absicht, welche Metriken wozu ausgelesen werden, kann unterschiedliche Ausmaße annehmen. Dies kann auf routinemäßiger Programmkontrolle, auf bessere Entwicklung, Fehleranalyse oder Anderes zurückführen. Es gibt bestimmte Metriken, welche bereits sehr weit etabliert sind, darunter fallen grundlegende Metriken wie das bereits angesprochene $\#LOC$, aber auch erweiterte Metriken wie die *Test Coverage*, *Functional Size* oder auch *Design Pattern Use* sind oft genutzt.

Die Metriken bilden für den Nachweis von Wert durch Struktur eine wesentliche Rolle. In den folgenden Abschnitten soll geklärt werden, worin man die Metriken unterteilt, wie man sie benutzen kann und zusammensetzt, sowie gegebenenfalls neue Metriken baut.

3.4.1 Software Produkt Attribute

Um ein System nach bestimmten Gesichtspunkten einzuschätzen, weiterzuentwickeln oder zu korrigieren, müssen diverse Eigenschaften am System eingeschätzt werden, welche sich je nach gewählten Gesichtspunkt unterscheiden. Diese System-Eigenschaften, genannt *Produktattribute*, sind allgemein gehalten sowie teilweise sehr abstrakt definiert. Ein Produktattribut ist dabei eine Vorstufe der Metriken, da diese als Attribut noch keine genaue Methode der Messung beinhalten. Hat man jedoch ein Produktattribut ausgewählt, können darauf beruhende Metriken genutzt werden, um dieses Attribut zu quantifizieren.

Nach Fenton *et al.* [16] werden Produktattribute nach der Klassifikation *Intern* und *Extern* unterteilt. Intern und Extern sind dabei wie folgt beschrieben:

1. *Externe* Attribute sind abhängig sowohl von dem Verhalten des Produkts sowie seiner Umgebung. Ein Beispiel wäre hier das Attribut „Zuverlässigkeit“, welches sowohl von der Plattform, der Konfiguration und der operationalen Umgebung abhängt. Ein weiteres Beispiel wäre die Wartbarkeit, welche nicht nur vom Code abhängt, sondern ebenfalls vom User und seinen verfügbaren Tools. Für alle externen Attribute gilt es, geeignete Metriken zu finden, welche diese Umstände mit einbeziehen, da ansonsten das Resultat und die Aussagekraft der Metrik verfälscht wird. In diese Kategorie fallen damit fast alle Qualitätsattribute von Software, wie *Security*, *Performance* und *Usability*, aber auch *Agility* und *Resilience*.

2. *Interne* Attribute sind diejenigen, welche rein vom Code oder den internen Strukturen abhängen. Das prominenteste Beispiel ist auch hier wieder die *Size* des System (ohne dabei direkt die Metrik *#LOC* zu benutzen, das wäre nur eine zugehörige Metrik dazu), aber auch *Code Complexity*, *Coupling*, *Cohesion* und *Modularity* sind interne Attribute, welche nur von der Code-Basis abhängen. Ein internes Attribut kann dabei durchaus für externe Attribute weiterverwendet werden.

Der nächste Schritt, nach der Auswahl der benötigten Attribute, ist die Festlegung der Metrik für diese Attribute. Dabei kann es für ein Attribut durchaus mehrere Metriken geben, zum Beispiel für *Complexity* nicht nur die Anzahl aller Verzweigungen pro Klasse, sondern auch *#GoTo* als Anzahl aller `goto`-Befehle pro Klasse im System. In den meisten Fällen wird eine Metrik für ein externes Attribut schwieriger zu gestalten sein als ein internes Attribut, da bei externen Attributen deren Umgebung mit einbezogen werden muss, was durchaus nicht so einfach zu erfassen ist wie die Anzahl aller Befehle im System. Dadurch sind meistens auch die internen Attribute einfacher maschinell erfassbar als die externen.

Ferner spielt bei der Auswahl einer geeigneten Metrik nicht nur das entsprechende nachzuweisende Attribut eine Rolle, sondern auch das Ziel, der Zweck und das Einsatzgebiet, der sogenannte *Scope* des Attributs oder der Metrik. In Fenton *et al.* [16] sind folgende Scopes von Attributen und Metriken genannt:

1. *Cost and effort estimation* (Aufwandsschätzung)
2. *Data collection* (Datenerfassung)
3. *Quality models and measures* (Qualitätserfassung)
4. *Reliability* (Zuverlässigkeitsprüfung)
5. *Security* (Sicherheitsprüfung)

6. *Structural and complexity metrics* (Komplexitätserfassung)
7. *Capability maturity assessment* (Qualität des Softwareentwicklungsprozesses)
8. *Management by metrics* (Management)
9. *Evolution of methods and tools* (Weiterentwicklung)

Der Scope spielt dabei eine Rolle bei der Entscheidung, welches Attribut erfasst wird. Dahingehend können Metriken und Attribute für mehrere Scopes genutzt werden, zum Beispiel dient das Attribut *Agility* mindestens für die beiden Scopes *Quality models and measures* und *Evolution of methods and tools*. Jedoch kann sich die Art der Auswertung der *Agility* über jeweils verschiedene Metriken unterscheiden.

Zusammengefasst kann man sagen, dass sich eine Metrik durch das nachzuweisende Attribut sowie den Scope definiert.

$$Metric = Attribute \times Scope$$

Beide Bereiche haben dabei erheblichen Einfluss darauf, inwieweit die resultierende Metrik erfasst wird und welche Aussagekraft sie besitzt.

3.4.2 Messen von Metriken im System

Das Erfassen einer Metrik im System ist ein durchaus komplexer Vorgang, je nachdem, welche Eigenschaft nachgewiesen werden soll. Dabei ist es entscheidend, welcher Scope und welches Attribut diese Metrik beeinflussen. Ist dieser Rahmen einmal gesetzt und eine entsprechende konkrete Metrik definiert, dann liegt es an der Umsetzung, die Metrik korrekt aus dem System zu extrahieren. Je nach gewählter Metrik kann dies die breite Spanne von „sehr einfach“ bis „fast unmöglich“ einnehmen, wobei man bei einer Metrik darauf bedacht ist, diese auch in der Praxis nutzen zu können, weswegen sich die Schwierigkeit des Auslesens meist in Grenzen halten wird.

In Fenton *et al.* [16] wird das Erfassen einer Metrik in zwei verschiedenen Kategorien unterschieden:

1. *Direct Measurement* ist die Erfassung von Metriken, welche sich direkt aus einem Zustand des Systems auslesen lassen. Sie sind meist atomar und immer unabhängig von anderen Metriken. Das beste Beispiel ist das bereits vielerorts verwendete $\#LOC$, da die Anzahl aller Codezeilen im Programm nicht von anderen Metriken zusammengesetzt ist und sich direkt auslesen lässt.

2. *Derived Measurement* von Metriken ist dagegen, wenn eine Metrik durch andere Metriken definiert wird. Ein Beispiel aus der Physik wäre hier die Dichte eines Objektes, welche sich zusammensetzt aus

   $$Dichte = \frac{Masse}{Volumen}$$

   Es wird nur schwer möglich sein, die Dichte direkt aus einem Gegenstand zu erfassen, ohne dessen Masse und Volumen zu messen. Dabei sind alle drei Komponenten jeweils eigene Metriken. Ähnlich verhält es sich in Softwaresystemen mit abgeleiteten Metriken, die wiederum nur aus direkten Metriken entstehen. Im Bezug auf die Metrik $\#LOC$ wäre eine entsprechend davon abgeleitete Metrik die *Error Rate*, welche sich aus

   $$Error\ Rate = \frac{\#Errors}{\#LOC}$$

zusammensetzt und damit die durchschnittliche Anzahl an Fehlern pro Code-Zeile bestimmt.

In fortgeschrittener Benutzung von Metriken wird auf abgeleitete Metriken kein Verzicht mehr sein, da sie meist die interessanteren Aussagen liefern. Auch in dieser Arbeit wurde fast ausschließlich auf abgeleitete Metriken gesetzt, da die Grenzen und Möglichkeiten der direkten Metriken sehr eingeschränkt sind und nur mit abgeleiteten Metriken das Ziel erreicht werden kann.

Die Art und Weise, wie genau man eine Metrik definiert, hängt zwar wesentlich von den Attributen und dem Scope ab, aber auch die Metriken selbst können auf unterschiedliche Weise ausgelegt werden. In Fenton *et al.* [16] wird das anhand der Zeilenanzahl $\#LOC$ aufgezeigt. Dabei kann diese Metrik („Wie viele Zeilen beinhaltet der Code?“) folgende Ausprägungen besitzen:

a) die $\#LOC$ beschreibt alle Zeilen im Code, egal was sie beinhalten,

b) die $\#NCLOC$ ($NC = non\ commented$) beschreibt all jene Zeilen im Code, welche kein Kommentar darstellen, und

c) die $\#CLOC$ beinhaltet dagegen ausschließlich die Codezeilen.

Die Aufsplittung von LOC in $NCLOC$ und $CLOC$ indiziert also genau genommen, dass die eigentliche $\#LOC$ nicht direkt sondern auch abgeleitet ist, jedoch kann man die $\#LOC$ auch ohne vorheriger Bestimmung von $\#NCLOC$ und $\#CLOC$ aus dem System ermessen, was diese Metrik also wieder zu einer direkten Metrik werden lässt. Jedoch können die beiden anderen Metriken sinnvoll sein, zum Beispiel für die Metrik *Comment Density* im System mit

$$Comment\ Density = \frac{\#CLOC}{\#LOC}$$

also der durchschnittlichen Kommentarzeilen pro Code-Zeile. Ebenso wäre die Metrik der funktionalen Codezeilen *Effective Code* eine entsprechend abgeleitete Metrik aus $\#NCLOC$ dividiert durch $\#LOC$.

Für die meisten weit etablierten Metriken gibt es automatische Methoden, entweder von der Entwicklungsumgebung selbst oder Drittprogrammen, welche alle nötigen direkten Metriken auslesen und abgeleitete Metriken daraus berechnen, am Ende meist noch grafisch aufbereiten. Diese Metriken können anschließend weiter für ihren Zweck und dem gewählten Scope genutzt werden. Es kommt am Ende auf die Interpretationsweise der Metrik an, ob sie dann auch kausal hinreichend gewertet werden kann. In Fenton *et al.* [16] wird dazu das einfache Beispiel genannt, dass man mit einem IQ-Test zwar halbwegs die Intelligenz eines Menschen ermitteln kann, jedoch nicht die Intelligenz eines Hundes, was die IQ-Metrik jedoch nicht falsch werden lässt, sondern lediglich die Schlussfolgerung aus dieser Metrik, sollte man einen IQ-Test an einem Hund durchführen.

3.4.3 Bauen von Metriken

Wie bereits in den vorherigen Kapiteln erwähnt, hängt eine Metrik von ihrem Scope und Attribut ab. Sollte für einen bestimmten Bereich noch keine geeignete Metrik vorliegen, gibt es die Möglichkeit, selbst eine Metrik für diesen Zweck zu erschaffen. Dies ist jedoch ein recht komplexer Vorgang, da man nicht einfach andere Metriken aneinander ketten kann. Man sollte

sich vorher bewusst sein, welche Faktoren in eine Metrik einfließen sollen und können, inwieweit der Zusammenhang dazwischen besteht, welche Auswirkungen eine Metrik bei ungewöhnlichen Inputs besitzt und ob das Ergebnis auch tatsächlich aussagekräftig ist. Folgender Prozess bietet sich an, bei dem Bau einer Metrik zu verfolgen:

1. Effekt oder Eigenschaft erfassen
2. Einflüsse auswerten
3. Abschätzung
4. Formale Repräsentation
5. Erfassung der Grenzen
6. Validierung

Diese sechs Schritte sollten idealerweise bei einer Neuentwicklung durchlaufen werden, um eine neue, belastbare Metrik auf das System anzuwenden. Sollte hierbei etwas ausgelassen werden, besteht die Gefahr, dass die resultierende Metrik komplett ungeeignet für gewählten Scope und Attribut ist. Im folgenden sollen kurz alle Schritte einmal beschrieben werden.

Punkt 1 „Effekt oder Eigenschaft erfassen“ beinhaltet zuerst die Erkenntnis, dass für eine bestimmte Fragestellung keine Metrik existiert. Dabei ist zu beachten, dass zwar eine entsprechende Metrik etabliert sein kann, jedoch einen anderen Scope oder ein anderes Attribut anspricht, was diese Metrik eventuell ebenso unnütz für das eigene Problem dastehen lässt. Hat man aber bereits eine Vorstellung über das eigene Problem und deren Auswirkungen, beziehungsweise weiß man bereits, welche Eigenschaft des Systems dafür infrage kommen könnte, ist der erste Schritt bereits getan. Meistens ergibt sich die erforderliche Eigenschaft direkt aus dem Problem selbst, wie zum Beispiel die Frage nach den effektiv funktionalen Teilen im Programm.

Punkt 2 „Einflüsse auswerten“ folgt daraufhin mit den Einflüssen, welche Auswirkungen auf die Ausprägung des Problems besitzen. Es benötigt einiges an Umsicht und Bekanntheit des Problems, um wirklich alle wichtigen Einflüsse abzudecken und jeweils auszuwerten. Hier werden erstmals die Kernpunkte der Darstellung bedacht.

Punkt 3 „Abschätzung“ erfordert die Überlegung, inwieweit die ausgemachten Faktoren ihren Einfluss ausüben, insbesondere deren relativen Einfluss zueinander. Dies kann sich entweder aus mathematischer oder empirischer Natur ergeben. Hat man die Kernpunkte der Darstellung erfasst, ist es einfacher, deren Beziehungen zueinander herzustellen. Dieser Schritt kann durchaus als Hauptarbeitsschritt bezeichnet werden, da es die Essenz einer Metrik sein wird, die Kernpunkte des darunterliegenden Modells der Darstellung logisch zusammenzufügen, um die Belastbarkeit der Metrik zu untermauern. Sobald die Zusammenhänge gefunden sind, sind es nur noch „Abrundungen“, welche die Metrik fertigstellen werden.

Punkt 4 „Formale Repräsentation“ erfordert eine formale Darstellung der Abschätzung. Dazu gehört einerseits die Repräsentation der einzelnen Faktoren, sowie deren Zusammenhang. In diesem Punkt liegt viel Aufwand, da man meist mit dem Problem und dessen Einflüssen zwar nun vertraut ist, jedoch nicht recht sicher sein kann, inwieweit man diese als formale Repräsentation aufstellt, damit alles einen Sinn ergibt. Es wird hierbei geregelt, welche Repräsentation die im vorherigen Schritt herausgefundenen Zusammenhänge haben.

Punkt 5 „Erfassung der Grenzen“ beinhaltet lediglich, dass man sich veranschaulicht, inwieweit die ermittelte formale Repräsentation an den Extremstellen Sinn macht. Dafür ist es meist geeignet, verschiedene Einflussfaktoren jeweils auf 0% sowie 100% zu setzen, nur um zu schauen,

wie sich die entwickelte Formel verhält. Eine Metrik ergibt nur dann Sinn, wenn alle Werte aus dem Funktionsbereich sinnvoll wiedergegeben werden können, und nicht etwa nur der Durchschnitt brauchbare Ergebnisse liefert. Man kann zwar bei dem Entwickeln und späteren Einsatz einer Metrik bestimmte Bereiche oder einzelne Funktionswerte ausschließen (zum Beispiel bei dem Wert 0 oder bei negativen Werten), bei denen die Metrik nicht angewendet werden kann, doch schmälert das mathematisch unbegründete Ausschließen von Funktionswerten die Relevanz der Metrik stark.

Punkt 6 „Validierung“ beinhaltet den Nachweis, dass die erschaffene Metrik auch dem Attribut und dem Scope dient. Nur weil in Punkt 5 die Grenzen korrekt ausgewertet wurden, bedeutet dies nicht, dass das Resultat der Metrik auch Sinn ergibt. Validierung kann dabei entweder durch empirische Messungen oder durch mathematische Bestätigung erfolgen. Hier werden sich unbeachtete Einflussfaktoren oder falsch gewertete Zusammenhänge aus den Schritten 2 und 3 spätestens durch eine fehlerhafte Metrik auswirken.

Bei der Zusammensetzung einer neuen Metrik, beziehungsweise bei der Erfassung aller Einflussfaktoren, kann man durchaus auf bereits etablierte Metriken zurückgreifen, da man davon ausgehen kann, dass diese bereits für bestimmte Zwecke entworfen und validiert wurden, solange sie dem gleichen eigenen Attribut und Scope dienen.

4 Redundanz

Es gibt viele Eigenschaften im System, die Einfluss auf den Wert und die Ausprägung des Systems haben. Einer davon ist die Redundanz im System, mit der sich diese Arbeit beschäftigt. Es kann viele Schadenspotenziale im System geben und die Redundanz ist nur eines davon. Der Nachweis, dass Redundanzen als Strukturelement im System Einfluss auf den Wert besitzen, erfordert hier allerdings ein eigenes Kapitel zur Thematik Redundanz, inklusive der Ursachen, Wirkungen, Typen und weiteres, damit anschließend kausale Zusammenhänge erklärt und herangezogen werden können.

Die Redundanz ist ein wichtiger Faktor, den es bei der Planung, Umsetzung und Weiterentwicklung eines Softwaresystems zu beachten gilt. Sie ist eine wichtige Systemeigenschaft, neben anderen wichtigen Systemeigenschaften wie *Cohesion* oder *Coupling*, welche durchgängig beachtet werden muss und dadurch in jeder Etappe im Lebenszyklus eines Systems Auswirkungen auf das Resultat besitzt. Das bedeutet jedoch nicht, dass Redundanz in jedem Falle negativ auftritt, denn durch gewollte Redundanz kann ebenso ein höheres Level an Stabilität und Sicherheit gewährleistet werden, aber durch ungewollte Redundanz kann dies wiederum nachteilig auf beides wirken. Die folgenden Abschnitte sollen das Phänomen der Redundanz weitläufig beleuchten und einen Einblick darin geben, was Redundanz im System bedeutet, inwieweit diese genutzt werden kann, wie sich Redundanz als Fehler einschleicht und welche Auswirkungen verschiedene Arten von Redundanz haben können.

4.1 Redundanz im System

Eine Redundanz bedeutet in erster Linie, dass mehrere Teile des Systems zum Beispiel eine gleiche Funktion, Aufgabe oder Intention abdecken. Ein solches „Duplikat“ kann alle möglichen Elemente eines Systems betreffen, von Quellcode-Ausschnitten bis hin zu Systemanforderungen. Jedoch ist der Begriff „Duplikat“ hierbei nicht wirklich angemessen, da dies eine Verdoppelung impliziert, welche das Phänomen der Redundanz nur spärlich beschreibt. Eine Redundanz ist dabei nicht auf zwei Versionen beschränkt sondern kann vielfach Auftreten und auch nur teilweise und überlappend andere Elemente kopieren. Das deutsche Wörterbuch *Duden* definiert Redundanz [15] folgendermaßen:

(1) *das Vorhandensein von eigentlich überflüssigen, für die Information nicht notwendigen Elementen;*

(2) *Überladung mit Merkmalen*

Das für diese Arbeit benötigte Verständnis von Redundanz ist dabei Punkt (1) der Definition, also Redundanz als Bezeichner für Daten ohne echte Mehrinformationen.

Dabei ist der wichtige Punkt bei der Einschätzung der Redundanz, zu unterscheiden, ob der mehrfach auftretende Teil vom System eine notwendiges Redundanz darstellt oder nicht. Der zweite Punkt wäre, ob diese Redundanz bewusst synchronisiert wird oder nicht. Das ist ein erheblicher Faktor, da eine nicht synchronisierte Redundanz schnell zu mehr Komplexität

Usage \ Managing	**Managed R.**	**Unmanaged R.**
Known/Wanted	YES	NO
Unknown/Unwanted	?	NO

Tabelle 4.1: Die verschiedenen (zulässigen) Ausprägungen von Redundanz

und damit weniger Flexibilität führt [19]. Dies alles besitzt damit immense Auswirkungen auf den Wert des Systems, welches sich unter anderem durch *Agility* definiert, was durch nicht geführte Redundanz negativ belastet wird. Die entsprechenden Kategorien sind aus Furrer [19] entnommen (basierend auf dem Konzept von Fowler [18]) und nochmals in Tabelle 4.1 als grafische Übersicht aufgezeigt.

Im Folgenden werden die dafür in der Fachwelt eingeführten Begriffe *managed* und *unmanaged Redundancy* benutzt, welche entweder auf eine „geregelte“ oder „ungeregelte“ Redundanz hinweisen. *Unmanaged Redundancies* können auf verschiedenste Wege in das System gelangen. Der erste Punkt ist bereits beim Definieren der Systemanforderungen. Dabei können die ersten Kopien in den Anforderungen auftreten, welche zum Beispiel durch verschiedene zuständige Personen eingeführt wurden, die jeweils unabhängig voneinander agieren. Auch durch eine unterschiedliche Vorstellung von Begriffen, Semantiken und Definitionen des Systems können verschiedenste Kopien entstehen, hierzu als Beispiel zwei Definitionen von dem Objekt „Kunden-ID“ als Semantik-Redundanz. Die *unmanaged Redundancy* an sich ist ein Faktor der *Technical Debt* und wird auch konstant als solche gehandhabt. Sollten *unmanaged Redundancies* einen Weg in das System finden, beabsichtigt oder nicht, trägt das Handeln die vollen Konsequenzen der *Technical Debt*, von den Folgen der erhöhten *TtM* und *DevC* bis hin zu nicht mehr wartbaren Systemen. Aber auch die weiteren Probleme der *Technical Debt* treffen auf die Redundanz zu, allen voran das schwierige Auffinden und die anschließende Beseitigung.

Die Ursache für *unmanaged Redundancy* ist häufig die mangelnde Kenntnis von dem zugehörigen Pendant, wie zum Beispiel die Nicht-Kenntnis von einer gleichartigen Funktion. Die unbeabsichtigte *unmanaged Redundancy* ist noch schwieriger zu identifizieren als die beabsichtigte, da hierbei das Auftreten sehr wahrscheinlich unbemerkt bleibt. Die Ursache für beabsichtige *unmanaged Redundancy* ist dahingehend ein klassischer Fall von bewusster *Technical Debt*, wo durch Zeitdruck, mangelnde Fachkenntnis oder begrenzte verfügbare Geldmittel Grenzen in der Entwicklung und/oder Evolution gesetzt sind. Sollte die dadurch aufgebaute „Schuld“ nicht schnellstmöglich abgebaut werden, dann äußert sich diese anschließend wie in Kapitel 2.3 „Fehler durch Architektur und Management“ durch eingeschränkte Möglichkeiten von Erweiterungen des Systems sowie vermehrten Aufwand in der Wartung bis hin zu einem *Fossil System*.

Nach Furrer [19] gibt es vielerlei Arten von Redundanzen im System, die häufigsten und meist genutzten Redundanzen sind jedoch die folgenden:

1. *Datenredundanz* ist die Kopie von Daten, welche in verschiedenen Elementen im System beherbergt sind. Somit gibt es für die gleichen Daten unterschiedliche Quellen, aus denen man diese auslesen und bearbeiten kann.

2. *Funktionale-Redundanz* bedeutet, dass funktionale Eigenschaften im System mehrfach vorkommen. Dabei spielt es keine Rolle, in welcher Art und Weise diese implementiert sind, da die Redundanz sich hier auf die Funktion des Elements bezieht, also zum Beispiel der Funktionsweise einer Methode.

3. *Interface-Redundanz* beinhaltet mehrfaches Auftreten von Interfaces mit der (teilweise) gleichen Funktionalität bezüglich der gleichen Elemente im System.

4. *Code-Redundanz* ist die klassischste Form der Redundanz in Software. Es bedeutet die exakte oder zumindest ähnliche Kopie von Code-Ausschnitten innerhalb des Systems ohne funktionalen Unterschied.

5. *Spezifikationsredundanz* ist ein mehrfaches Auftreten von Anforderungen und Spezifikationen an das System.

Neben den oben genannten Redundanzen in Software gibt es noch weitere Typen, welche jedoch meist weniger Beachtung finden. Im Prinzip kann jedes Element im Softwareentwicklungsprozess einen redundanten Gegenpart besitzen. Viele Redundanzen werden entweder durch Nicht-Wissen der Redundanz in Form einer Erweiterung eingebaut. Eine Interface-Redundanz entsteht zum Beispiel am häufigsten durch eine neue Anforderung, zu der alte verfügbare Interfaces nicht ausreichen. Eine neue Schnittstelle soll Abhilfe leisten, übernimmt aber nebenher noch Aufgaben des alten Interfaces. Anstatt also das alte Interface zu erweitern, wurde bei dem Bau einer neuen Schnittstelle eine Interface-Redundanz geschaffen. Anders verhält es sich bei Code-Redundanz, bei welcher anstatt (geteilter und parametrierbarer) Funktionalität aus Code-Bibliotheken zu beziehen sich auf eine reine Kopie des Codes aus anderen Elementen beschränkt wird. Das ist besonders dann kritisch, wenn im ursprünglichen Code Fehler behoben werden oder zum Beispiel die Performance verbessert wird, die Code-Kopie jedoch nicht beachtet wird, da diese entweder nicht (mehr) bekannt ist oder anderweitig eingebunden ist, sodass sich die Redundanz nicht einfach entfernen lässt.

Da es in dieser Arbeit hauptsächlich um die Datenredundanz geht, soll hier noch einmal etwas näher darauf eingegangen werden. Datenredundanz entsteht meist dadurch, dass für den Bau neuer Funktionalitäten oder bei der Fehlerbehebung externe und entfernte Daten benötigt werden. Als Beispiel soll hier der Bedarf an Kundendaten für eine neue Funktion im System dienen. Die positive Variante zur Vermeidung von Datenredundanz wäre, ein entsprechendes Interface der externen Kundendaten zu nutzen oder zu kreieren, anschließend dieses Interface zur Interaktion mit den Daten zu nutzen. Benötigt man die Daten jedoch lokal (zum Beispiel aus Performance- oder Sicherheitsgründen), wird eine synchronisierte Kopie (*managed Redundancy*) von den Originaldaten erstellt, welche jederzeit hinreichend die Daten mit der originalen Quelle abgleicht. Ein negativer Weg wäre der, sich selbst durch die neue Funktion einen eigenen (neuen) Datenpool zu schaffen, darin die kopierten Werte abzulegen und bei Bedarf vorher abzugleichen, ob sich etwas geändert hat. Das klingt eventuell auf den ersten Blick plausibel, jedoch beginnen dann die Probleme, wenn andere Funktionalitäten den neuen Datenpool nutzen, ohne dass dieser vorher abgeglichen wurde. Oder der originale Kunden-Datenbestand ändert sich und der Abgleich der neuen Funktion ist unzureichend, weil zum Beispiel neue Datenbank-Felder im Originalbestand nicht erfasst werden. Das Ganze führt dann zu Unterschieden in den Daten, was sich wiederum in Problemen im operativen Einsatz äußern kann, was zu Ausfällen des Systems oder Schadensersatz an Dritte führen kann, was bei großen Systemen meist sehr teuer wird. Dies wäre dann eine nicht hinreichend synchronisierte Redundanz, eine sogenannte *unmanaged Redundancy*. Ist die *unmanaged* Datenredundanz einmal im System, dann ist es nach längerem Zeitraum sehr schwer, diese wieder aufzufinden und zu entfernen. Ein etwas extremeres, dafür umso häufiger vorkommendes Beispiel für *unmanaged Redundancy* sind aus einem Datenbestand kopierte Werte, welche dann im Quellcode „hart einprogrammiert“ für verschiedene Operationen benutzt werden, zum Beispiel für die Filterung einer bestimmten ID. Das Problem hieran wäre, dass sich die eigentliche ID im originalen Datenbestand ändern könnte, die „hart“ hinein

geschriebene ID (also die redundante Kopie) jedoch nicht synchronisiert erhalten bleibt und damit Fehler im System verursacht. Wie bei *Technical Debt* allgemein gültig, ist es sinnvoll, die entstandene „Schuld“ schnellstmöglich wieder abzubauen. Jedoch ist dies nur dann möglich, wenn die entstandene Redundanz auch bekannt ist.

4.1.1 Managed Redundancy

Die *Managed Redundancy* (geregelte Redundanz) ist ein Ansatz, notwendige Redundanz unschädlich zu machen. Die Gründe, warum eine Redundanz notwendig ist, sind vielfältig: Performance, geografische Verteilung, Disaster Recovery, Hochverfügbarkeit, Load Balancing und Sicherheit sind nur einige der möglichen Szenarien der notwendigen Redundanz [19]. Aber auch externe Faktoren können notwendige Redundanz verursachen, wie zum Beispiel die Nutzung von Third-Party-Software im System oder die Migration von Teilen des Systems [41].

Für die Handhabung der notwendigen Redundanz gibt es ebenfalls ein entsprechendes Architektur-Prinzip, wie es allgemein in Kapitel 2.2.3 „Architekturprinzipien“ vorgestellt wurde. Dieses besagt folgendes:
„There is exactly one source for the functionality and for the data (both during development time and during runtime). All redundant copies must be materially and time-wise synchronised (also partial copies).“ [19], Satz 4.
Dies besagt, dass es in jedem Falle nur eine Quelle von Daten und Funktionalität geben soll, und alle notwendigen Redundanzen müssen hinreichend gut synchronisiert werden. Die Gründe für notwendige Redundanz wurden bereits behandelt. Daraus ergibt sich die Frage danach, was eine hinreichend gute Synchronisation bedeutet. Für sehr große Systeme bietet sich ein globales Management von Redundanzen an [41]. Aber auch die Nutzung von einzelnen dedizierten Funktionalitäten für die Synchronisation wäre möglich. Letzten Endes ist es immer möglich, pro Redundanz-Verbindung eine Synchronisation einzurichten, welche bei Veränderungen triggert. Um jedoch zu großen Overhead durch Synchronisation zu vermeiden, sollte generell Redundanz vermieden oder zumindest eine strategisch einfache Lösung für solche Fälle gefunden werden. Was genau „hinreichend“ bedeutet, das hängt von der Quelle der Daten oder Funktion ab. Bei einem Tages-Backup reicht es, alle 24 Stunden eine Synchronisation anzustreben, bei Kunden-Transaktionen sollten jedoch alle Änderungen schnellstmöglich an alle notwendigen Kopien gesandt werden, wobei der Zeitversatz dafür mit einbezogen werden sollte.

Eine geringe und geregelte Redundanz im System sichert eine Agilität und Flexibilität, welche bestmöglich erhalten werden sollte. Jede nicht geregelte Redundanz führt unweigerlich zu Verlust am Wert des Systems, wofür diese Arbeit auch den kausalen und quantifizierten Nachweis bringt.

4.1.2 Unmanaged Redundancy

Ganz im Gegensatz zu der geregelten Redundanz für notwendige Fälle gibt es die ungeregelte Redundanz, die *Unmanaged Redundancy*, welche immer dann auftritt, wenn eine Redundanz im System entgegen des Redundanz-Prinzips nicht hinreichend Synchronisation erfährt. Der entscheidende Unterschied hierbei liegt nicht an der nicht vorhandenen Notwendigkeit der Redundanz, sondern einzig allein an der nicht sauber implementierten Synchronisation beider Elemente. Ein gutes System ist idealerweise frei von ungeregelter Redundanz [41].

Die Ursachen für Redundanz wurden bereits weiter oben aufgeführt, ebenso die Ursachen für eine Implementierung von Redundanz ohne Beachtung des Architekturprinzips und ohne hinreichend guter Synchronisation. Zeit, Erfahrung oder Budget zwingen mitunter dazu, eine lose Kopie im System anzufertigen, ohne weiterführenden Blick auf die Zukunftsfähigkeit des

Systems. Dabei kann sich der Wert des Systems durch diesen „schnellen Weg“ temporär steigern, da die beiden wichtigen Faktoren *TtM* und *DevC* dafür sinken. Das ist leider auch der häufigste Grund dafür, dass solche unsauberen Lösungen nicht direkt gefunden werden oder gar im System behalten werden, ungeachtet der eigentlichen aufgebauten und zu beseitigenden „Schuld“. Das eigentliche Problem daran ist aber nicht nur Prinzip-bedingt, sondern besitzt auch durch die Rückwirkung in Form von sinkender *Agility* materiellen Wertverlust. Aber auch wenn man sich der eingebauten ungeregelten Redundanz bewusst ist und diese später entfernen möchte, ist das ebenfalls einfacher gesagt als getan, da durchaus der Entwickler, Designer oder Verantwortliche für den Part nicht (mehr) verfügbar ist oder zu viele andere Personen mit der eingebauten Redundanz weitergearbeitet haben, sodass diese nicht mehr in der Ursprungsform vorliegt. Und selbst wenn man diese Hürde genommen hat und die Redundanz auffinden konnte, ist es damit noch nicht vorbei, da eventuell mittlerweile andere Komponenten von der Redundanz abhängig sind, sodass diese nicht einfach entfernt werden kann.

Der einfachste Weg, eine gefundene ungeregelte Redundanz zu entfernen, ist, diese in eine geregelte Redundanz zu wandeln. Dann ist die hinreichende Synchronisation gegeben und es besteht, aus der Sicht des Redundanz-Prinzips, keine Gefahr mehr. Allerdings sollte einem dann bewusst sein, dass dies auch Auswirkungen auf andere Bereiche besitzt. Neben dem Aufwand, den man in die Umwandlung steckt, muss eine stärkere Zunahme der Komplexität im System erwartet werden, was wiederum zukünftige Erweiterungen des Systems teuer werden und länger dauern lässt. Die einzige Lösung, welche ohne Nebenwirkungen auf das System auskommt, ist ein komplettes Refactoring der abhängigen Komponenten, was wiederum sehr teuer und aufwendig werden kann. Aber nur damit wäre sichergestellt, dass das System so funktioniert, wie es sollte.

Alles zusammen ergibt die Erkenntnis, dass eine ungeregelte Redundanz einer der größten Kosten- und Komplexitäts-Treiber in der Entwicklung von Software ist, was auch von der Literatur empirisch bestätigt wurde [41]. Um diese Kosten und die Komplexität zu vermeiden, ist der einfachste Weg, dass der federführende Softwarearchitekt in jeder Phase darauf besteht, das Prinzip der Redundanz einzuhalten. Es ist jedoch absehbar, dass der Architekt nicht alles daraufhin überprüfen kann, weswegen eine starke Einbindung von Entwicklern und Stakeholdern gleichermaßen erforderlich ist.

4.2 Auswirkungen und Folgen

Ist eine *unmanaged Redundancy* erst einmal eingebaut, entstehen verschiedene Bereiche, welche sich früher oder später negativ auf das System auswirken, was auch bereits im vorherigen Abschnitt angerissen wurde. Eine *unmanaged Redundancy* hat direkt oder indirekt Auswirkungen auf den Wert des Systems, aber insbesondere folgende Wert-bestimmenden Attribute des Systems:

1. *Qualität* besitzt in Softwaresystemen hohen Stellenwert, da das System direkten Einfluss auf das Business hat. Durch mangelnde Architektur und unsaubere Umsetzung der Prinzipien sinkt die Qualität durch vermehrt auftretende Fehler, nicht benutzbare Features oder es ergeben sich Nachteile gegenüber Konkurrenzprodukten.

2. *Maintenance* meint, dass durch die ungeregelte Redundanz erheblicher Mehraufwand für die Wartung des Systems betrieben werden muss. Da durch Wartung keine neuen Features eingebaut werden sondern nur das aktuelle System auf einen fehlerfreien Stand gebracht wird, ist ein Mehraufwand in der Wartung ein direkter Faktor für erhöhte Kosten und Zeit.

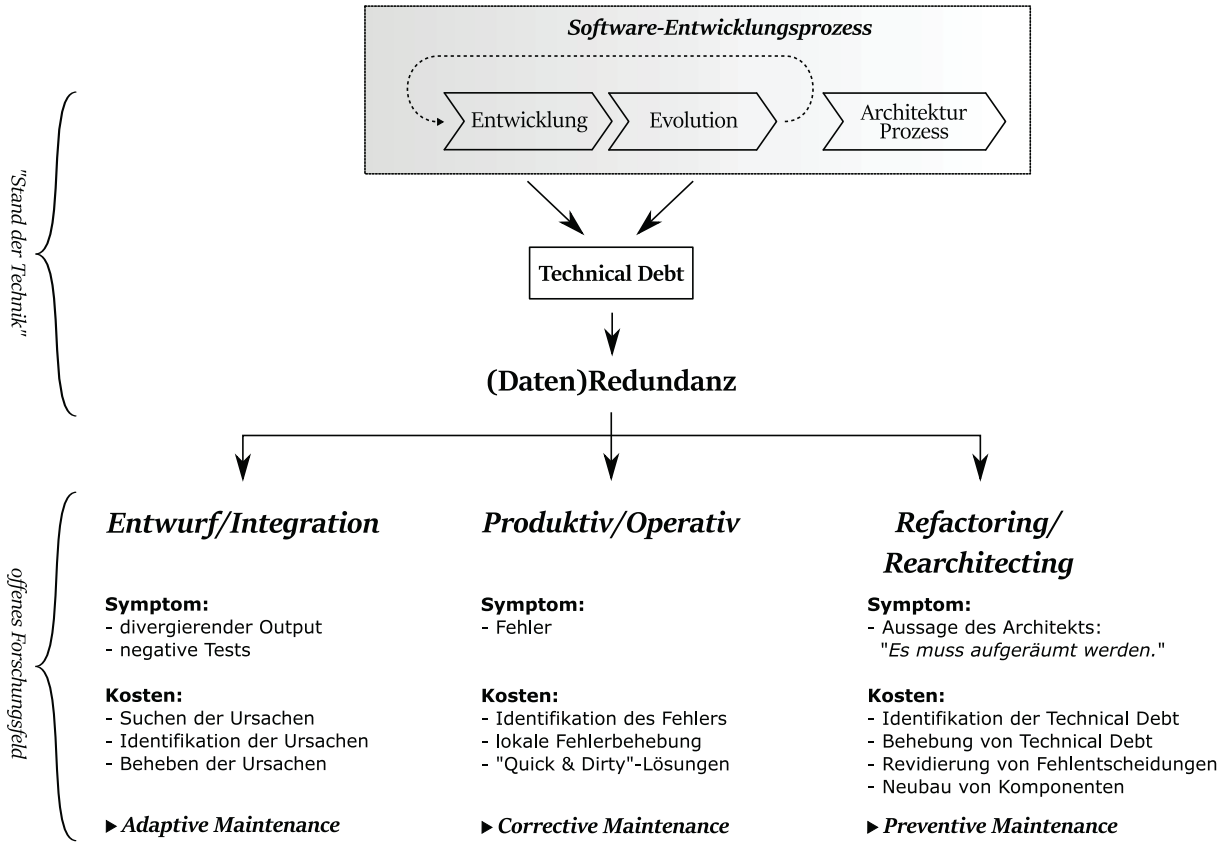


Abbildung 4.1: Die Einflüsse der verschiedenen Ursachen von Kosten

3. *Evolution*, das heißt die Weiterentwicklung des Systems, wird durch ungeregelte Redundanz insofern behindert, weil zusätzlicher Aufwand betrieben werden muss, um eventuelle Bugs zu fixen, welche durch die Redundanz auftreten, oder aber erneut Umwege geschaffen werden müssen, um das ursprüngliche Problem nicht schwerwiegender werden zu lassen.

Durch die hier aufgeführten Auswirkungen sinkt die Flexibilität des Systems, welche einen zentralen Bestandteil eines zukunftsfähigen Softwaresystems darstellt [41]. Da die *unmanaged Redundancy* einen Teil der *Technical Debt* darstellt, gelten auch für sie die drei Kostenbereiche, welche in Kapitel 2.1.2 „Die Kosten eines Systems“ beschrieben sind und den Wert des Systems definieren: *Adaptive*, *Corrective* und *Preventive Maintenance*. Diese drei Bereiche, welche sich teils direkt oder nur indirekt mit den oben genannten Wert-bestimmenden Attributen von Redundanz decken, sind demzufolge auch bei der Bestimmung von den Folgen *unmanaged Redundancy* zu beachten. Dabei setzt sich der Wertverfall durch diese *unmanaged Redundancy* zusammen aus Kosten der Evolution und Weiterentwicklung, dem Refactoring zum Beseitigen von fehlerhaften Strukturen sowie den operativ zur Laufzeit auftretende Fehlerbehebungen und Kosten für Ausfälle, Legal&Compliance-Angelegenheiten und weiterem. Die volle Fehleranalyse kann man in Abbildung 4.1 betrachten, wo dargelegt ist, wie die drei Kostenbereiche aus dem Softwareentwicklungsprozess entstehen. Es bleibt die Frage, wie man diese drei Bereiche bezüglich der Redundanz in ihrem Einfluss und ihrer Höhe ermittelt. Dafür wurden in dieser Arbeit die sogenannten „Wert-Metriken“ erstellt, welche später zum Einsatz kommen, um die Folgen von *unmanaged Redundancy* nachzuweisen.

In Lilienthal [34] wird genauer auf die (der Datenredundanz ähnlichen) Code-Redundanz

eingegangen und dabei werden die Folgen derer wie folgt beschrieben:

1. Duplikate erzeugen unnötig viel Komplexität,
2. Duplikate erhöhen den Aufwand für das Testen und Tests schreiben, und
3. die Wartung wird durch Duplikate erheblich erhöht, da mehrere Stellen analog geändert werden müssen.

Allerdings kann hierbei Tool-Support eingesetzt werden, um Code-Duplikate ausfindig zu machen. Dies unterscheidet Code-Redundanz von vielen anderen Redundanz-Arten, unter anderem auch von der Datenredundanz, wo identische Daten in verschiedenen Formen dargestellt werden können, sodass ein Auffinden sehr erschwert wird.

Das Finden und Entfernen von *unmanaged Redundancy* (besonders von *unmanaged* Datenredundanz) sind zwei Probleme, die viel Zeit und Aufwand bedeuten. Es hängt stark davon ab, wie lange die Entwicklung her ist, wie gut der Code dokumentiert ist und wie viel Fachpersonal dafür (noch) zur Verfügung steht. Zusätzlich sind beim Entfernen noch die Faktoren einzubeziehen, welche bestimmen, wie und wodurch die Redundanz beseitigt wird, da dies erheblichen Einfluss darauf hat, wie sauber die Lösung umgesetzt werden kann. All diese Informationen sind schwerlich zu erfassen und variieren von Unternehmen zu Unternehmen, weswegen diese auch in dieser Arbeit später noch als „Unternehmens-Faktoren“ in die Metriken eingebaut werden.

Eine *unmanaged Redundancy* im System zu finden, hängt stark davon ab, um welche Art von Redundanz es sich handelt. In Murer *et al.* [41] findet sich sehr oft der Hinweis, dass Modelle und Maps dabei weiterhelfen. So hilft eine *BCM* („Business Component Map“) dabei, funktionale Redundanz im System ausfindig zu machen. Tools können dabei helfen, exakte Code-Kopien als Redundanzen zu finden und Datenabgleich hilft gegen Daten-Redundanz. Aber auch hierbei gibt es wieder Probleme beim Auffinden, zum Beispiel könnte eine *unmanaged Redundancy* einer Datenbank andere Spaltennamen besitzen, was wiederum einen Inhalts-basierten Datenabgleich mit anschließender Analyse erforderlich machen würde. Noch schlimmer wäre eine *unmanaged Redundancy* einer Datenbank, in welcher die kopierten Daten selbst nicht mehr in einer Datenbank liegen, sondern in einem Text-Dokument. Das würde bedeuten, dass man vorher eine Code-Analyse durchführen muss, um überhaupt zu erkennen, dass die Kopie selbst keine Datenbank im zentralen System ist.

5 Der Nachweis

5.1 Problematik und Methodik

Das Phänomen, welches mithilfe dieser Arbeit kausal beschrieben werden soll, ist der bereits mehrmals genannte Wertverlust eines Softwaresystems durch *unmanaged* Datenredundanz. Wie in den vorherigen Kapiteln eingeführt, wird hierzu der strukturelle Wert einer Software betrachtet, die Darstellbarkeit der Redundanz als Graph und die anschließende Analyse mittels Metriken.

Die Grundlage für diesen kausalen Nachweises für die Theorie dieser Arbeit bildet die folgende Hypothese:

Hypothese
Es gibt einen kausalen und quantitativen Zusammenhang zwischen dem (Evolutions-)Wert der Software und deren Architekturprinzipien-Treue.

Um die Theorie im naturwissenschaftlichen Sinne zu bestätigen, bedarf es diverser Annahmen, Voraussetzungen, einer logischen Kette an Schlussfolgerungen und Belegen, damit sich letztendlich der Nachweis durch Nachvollziehbarkeit ergibt.

Die Thematik, welche diese Arbeit behandelt, ist einmal schematisch in Abbildung 5.1 aufgezeigt. Der absteigende „Arm“ ist bereits durchlaufen, welcher beinhaltet, wie die vorgenommene Spezifizierung der Architekturprinzipientreue ausgelegt wird: über den Einfluss der Prinzipien, die Auswirkungen der *Technical Debt*, die Funktionsweise von Redundanz und dem zugehörigen Architekturprinzip. Der zweite Part dieser Arbeit ist der aufsteigende „Arm“, der gesamte Nachweis, hin zu der bereits erwarteten und dadurch bestätigten Hypothese zurück, über die Darstellung (rote Begriffe) und die Messbarkeit (blaue Begriffe) von Graphen und Metriken, womit insgesamt die Theorie bestätigt wäre. Dieses Schema ist sozusagen die „Roadmap“ der Arbeit, und derzeit befindet man sich an der Kehrtwende. Der gesamte Nachweis (und damit die oben genannten Elemente) ist in dieser Arbeit in drei große Abschnitte eingeteilt:

1. *Das Graphmodell* ist der Teil des Nachweises, der die *Darstellung* in dem Nachweis bildet. Das Modell bildet damit die Grundlage aller kausaler Schlussfolgerungen und ist dementsprechend im ersten Block „I: Das Graphmodell“ (Kapitel 5.2) ausführlich behandelt.

2. *Die Metriken* setzten auf das Graphmodell auf und bilden die *Messbarkeit* im Nachweis. Die Metriken werden dafür benutzt, die Schlussfolgerungen zu ziehen, die schlussendlich den Wertverfall quantifizieren. Die Metriken werden im zweiten Block „II: Die Metriken“ (Kapitel 5.3) genauer erläutert.

3. *Der Wert des Systems* ist das Resultat des Nachweises und bildet das Ende der Beweiskette. Dies beinhaltet die Zusammenfassung sowie das Extrahieren des Wertes des Systems aus den Metriken. Dafür ist der dritte Block „III: Der Wert des Systems“ (Kapitel 5.4) vorhanden, um das Vorgehen genauer darzulegen und Resultate aufzuzeigen.

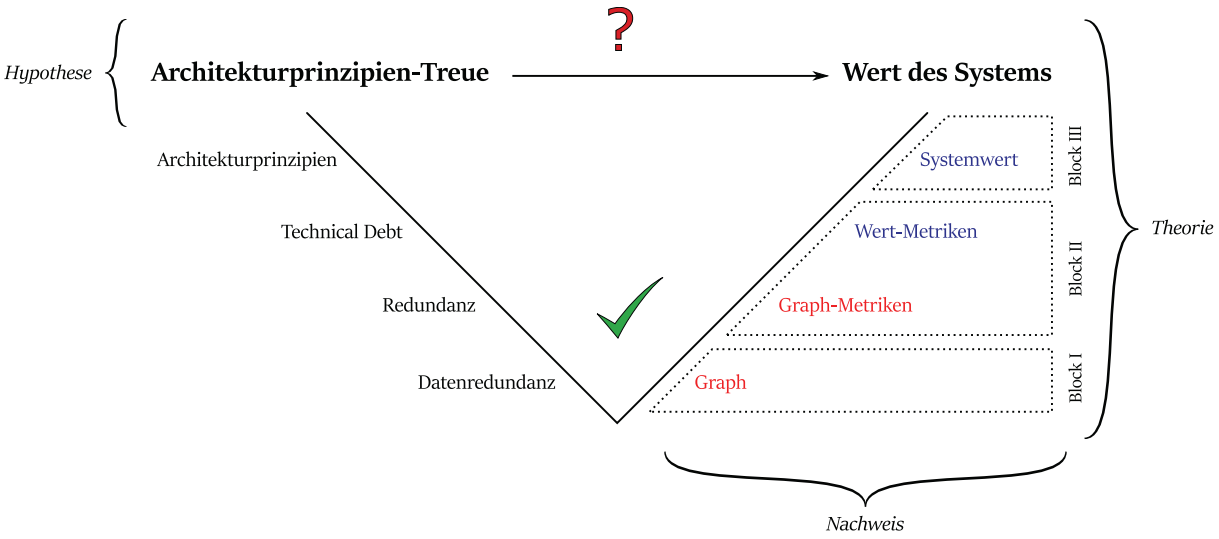


Abbildung 5.1: Das Vorgehen der Arbeit bezüglich Theorie, Hypothese und Nachweis

Dass ein Graph sich anbietet, um strukturelle Eigenschaften eines Systems zu erklären, wurde bereits in Kapitel 3.2 „Softwaresysteme als Graphen“ aufgezeigt. Der zu benutzende Graph wird mit seinen Knoten die Elemente des Systems widerspiegeln, dessen Kanten werden die Eigenschaft der Redundanz darstellen. Mithilfe einiger Eigenschaften des Graphen wird die *Darstellung* des Nachweises erreicht, da auf dem entstandenen Graphen formale Schritte vollführt werden können, die anschließend die *Messbarkeit* stützen. Diverse Metriken werden im Laufe des Nachweises eingeführt, zum einen für den Graphen selbst und dessen Auswertung, zum anderen auch Metriken für den Wert der Software, aufbauend auf den Metriken für den Graphen. Im Resultat werden alle erstellten Mittel zusammengeführt um eine eindeutige Aussage zu erhalten.

Die Schwierigkeiten, welche bei diesem Nachweis auftauchen, ist der Natur von Softwarearchitektur geschuldet. Es gibt nicht einmal eine einheitlich und allgemein geltende Definition von Softwarearchitektur und -design, weswegen ein kausaler Nachweis dieser beiden Themen recht schwer werden kann. Im Gegensatz zu der im Kapitel 1.2 „Motivation für den Nachweis von Werten der Architektur“ genannten Methode des naturwissenschaftlichen Beweises ist in dieser Arbeit kein praktischer Teil vorausgegangen. Das bedeutet, dass diese Arbeit den reinen theoretischen Teil betrachtet, welcher im Idealfall auf Erfahrungen und Beobachtungen aufbaut und logisch auf eine sinnvolle Theorie verweisen kann. Um das Argument der Allgemeingültigkeit des Nachweises zu erhalten, werden diverse Teile des Nachweises mit Variablen und veränderlichen Komponenten versehen sein, die bei Bedarf variiert werden können. Das ändert nichts an der Beweisführung und dem Resultat an sich, lediglich ist der Wert des Resultats nach einer Anpassung näher an der Beobachtung als vorher. Damit wäre eine stetige Annäherung der Beobachtung und Theorie auch im Nachhinein gegeben. Dass keine praktischen Beobachtungen im Voraus erfolgen konnten, ist dem zeitlichen Rahmen dieser Arbeit geschuldet, was der hier behandelten Theorie jedoch keinen Abstrich geben wird.

Wie in vorherigen Kapiteln erwähnt, ist der Kernpunkt dieser Arbeit der Nachweis von Datenredundanz im System sowie eine Feststellung des Wertes des Systems durch jene. Es gibt die verschiedensten Arten von Redundanz, jedoch benötigt jede Art ihre eigene Betrachtungsweise, auch wenn viele durch die hier angewendeten Methoden vermutlich ebenso quantifiziert werden könnten. Im Folgenden wird daher der Einfachheit halber im Zusammenhang mit dem Wort

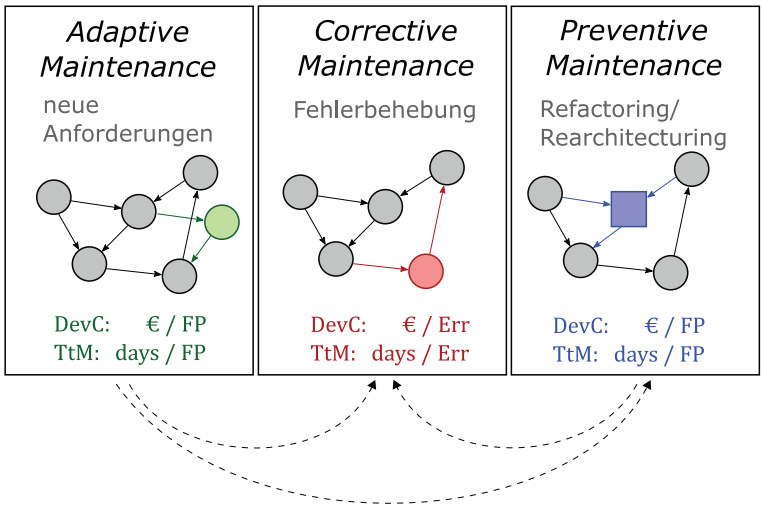


Abbildung 5.2: Die drei strukturellen Kosten/Wert-Bereiche

„Redundanz“ hauptsächlich die Datenredundanz gemeint sein, außer es wird explizit auf mehrere Redundanz-Arten verwiesen. Zudem wird hierbei entweder noch zusätzlich von *managed* oder *unmanaged* Datenredundanz gesprochen, einer entsprechenden Unterart der *managed* und *unmanaged Redundancy*.

Um den (strukturellen) Wert der Software nachzuweisen, sind die drei Struktur-Kostenbereiche wichtig, welche in Kapitel 2.1.3 „Der Wert eines Systems“ aufgeführt und noch einmal ausschnittsweise in Abbildung 5.2 abgebildet sind. Um den Wert zu bestimmen, werden die anfallenden Kosten genutzt, um den Wertverfall aufzuzeigen. Die Hypothese besagt, dass ein System mit *unmanaged* Datenredundanz dessen Wert mindert. Um das zu belegen, ist es notwendig, einen messbaren Verlust durch strukturelle Einschränkungen darzulegen. Die Methode, welche dafür hier gewählt wird, ist der Weg über die *Agility*, welche für sich selbst nicht nur als Eigenschaft für ein zukunftsfähiges System steht, sondern auch in den beiden Skalen *DevC* sowie *TtM* gemessen werden kann. Diese beiden Faktoren werden als Investition in das Produkt „Software“ betrachtet, welche entscheidend das Ausmaß an Kosten angibt. Der nächste Schritt in der Kette an Messwerten ist die Abschätzung der funktionalen Systemeigenschaften, welche in dieser Arbeit in *Function Points* (FP) oder *Use Case Points* (UCP) dargestellt werden. Diese Metriken sind bereits breit etabliert und geben einen ungefähren Überblick über die funktionale inhaltliche Größe der einzelnen System-Komponenten. Der Wertverfall äußert sich nun in einer Änderung an der Struktur der Software, deren Aufwand wiederum durch die oben genannten Maße eingeschätzt werden kann. Schlussendlich kann durch einen quantifizierten Mehraufwand gezeigt und eingeschätzt werden, inwieweit das System mehr Investitionen als nötig erfordert und damit als Produkt an Wert verliert.

Ab hier werden nun in Folge der logischen Nachvollziehbarkeit alle größeren Annahmen und Definitionen entsprechend markiert. Dabei unterscheiden sich diese beiden Punkte dadurch, dass eine ausgeschriebene Definition für den Nachweis essenziell und „nicht verhandelbar“ ist, ohne die ein Nachweis der Datenredundanz auf Basis der Architektur nicht möglich wäre. Die Annahmen hingegen stellen zwar auch einen essenziellen Teil dar, sind aber eher als logische Schlussfolgerung zu sehen, welche als grundlegende Axiome für den weiteren Nachweis benötigt werden, um darauf aufzubauen. Damit sind Annahmen (im Gegensatz zu Definitionen) durchaus abänderlich, sodass neue Wege in der Beweisführung begründet werden können.

5.2 I: Das Graphmodell

Wie bereits einige Male angesprochen, wird die Grundlage des Nachweises eine neu konstruierte Form eines strukturellen Graphen bilden. Dieses Kapitel soll den ersten relevanten Schritt in der Beweiskette bilden und die *Darstellung* repräsentieren.

Wie in Kapitel 3.1.1 „Darstellbarkeit“ erläutert, ist die Darstellbarkeit ein wichtiger Punkt um das eigentliche Problem zu beleuchten. Dies beinhaltet, das Problem nicht nur zu erfassen, sondern auch in einer geeigneten Form mit allen nötigen Eigenschaften abzubilden. Eine nicht erfasste Eigenschaft ist dahingehend fatal, weil dadurch eventuell ein Schritt in der Beweiskette nicht logisch vollzogen werden kann und der Beweis ins Stocken gerät und falsche Verläufe nimmt. Ebenso ist die gegenteilige Situation davon, eine überflüssige Eigenschaft in der Darstellung, eine unnötige Komplexität in der Beweiskette, welche durchaus das Potenzial besitzt, falsche Schlüsse in der Beweisführung zu unterstützen. Im Laufe der Entwicklung des Nachweises und der Quantifizierung wurden verschiedene Möglichkeiten gegeneinander abgewägt, darunter auch über die Aufnahme von diversen Eigenschaften und deren Ausprägung. Diese Entscheidungen werden, sofern sie relevant oder anderweitig interessant sind, auch an den entsprechenden Stellen aufgezeigt und möglichst nachvollziehbar begründet.

Durch den naturwissenschaftlichen Ansatz dieses Beweises ist auch die Darstellung als Graph hauptsächlich auf Beobachtung und Einschätzung aufgebaut. Das Ergebnis und die damit geschaffene Darstellung für das in der Hypothese angesprochene Problem ist ein *labeled, annotated, directed Multigraph*, welcher die notwendigen Eingenschaften besitzt, Redundanzen im System irgendwie strukturell darzustellen. Diese Art von Graph eignet sich besonders gut um diverse strukturelle Eigenschaften an Softwaresystemen darzustellen, immer mit der Option, diverse Eigenschaften des Systems abzubilden oder nicht. Die genannten Eigenschaften des Graphen bedeuten folgendes:

1. *labeled* ist die Eigenschaft des Graphen, die seine Elemente (Knoten und Kanten) eindeutig identifizierbar macht, sozusagen als „Name“ der Elemente. Dies wird für Knoten (Ecken) als v_i, beziehungsweise für Kanten als e_i dargestellt, jeweils als englische Bezeichner für *Vertices* und *Edges* mit dem Index i. Es ist essenziell, dass eine Identifikation via Labels eindeutig ist, damit keine Elemente doppelt vorkommen, selbst nach einer Abänderung oder Entfernung einiger Teile des Graphen und zugehöriger Eigenschaften. Ein Label ist dabei jedoch selbst keine Eigenschaft (Annotation), sondern tatsächlich nur eine „Identifikationsnummer“. Hierbei sei bereits darauf hingewiesen, dass das aus mathematischer Sicht benannte e_i für eine Kante nichts mit dem späteren e_i aus dem Nachweis zu tun hat, welches in dieser Arbeit die Bezeichnung für das „Element“ eines Systems darstellt. Um Verwirrung zu vermeiden, wird die mathematische Kante e_i hierfür ab nun in dieser Arbeit mit e'_i bezeichnet.

2. *annotated* ist der Graph genau dann, wenn seine Elemente (Knoten und Kanten) weitere innere Attribute und/oder eine Typisierung besitzen. Für Attribute oder Typ eignet sich im mathematischen Sinne die Bezeichnung $A(v_i) = A_{v_i} = n$ mit A als Attribut oder Typ, mit v_i als das entsprechende Element mit der Eigenschaft (hier Knoten, für eine Kante wäre es e'_i) und dem eigentlichen Wert des Attributs n.

3. *directed* meint, dass die Kanten zwischen Knoten durchaus verschiedene Richtungen aufweisen können, was die (einseitige) Beziehung zwischen Elementen darstellt. Dabei gibt es verschiedenste Möglichkeiten, dies formal darzustellen. Eine praktische Lösung wäre folgende Notation: $e'_i = (v_a, v_b)$ als ungerichtete Kante e'_i zwischen den Elementen v_a und v_b.

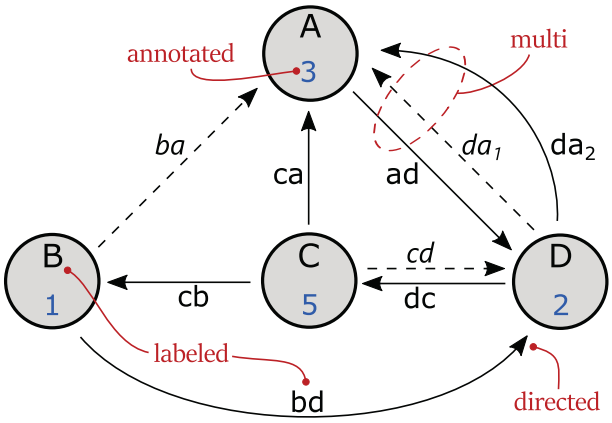


Abbildung 5.3: Ein zufälliges Beispiel eines *labeled, annotated, directed Multigraphen*

Eine gerichtete Kante könnte wie folgt aussehen: $e'_i = \langle v_a, v_b \rangle$ als Kante e'_i mit dem Anfang bei Element v_a und dem Ende (Pfeilkopf) bei v_b. Problematisch wäre es bei einer doppelt gerichteten Kante mit einer Pfeilspitze pro Ende. Das Problem kann man allerdings dadurch umgehen, dass eine doppelt gerichtete Kante mit einer ungerichteten Kante gleichzusetzen wäre.

4. *Multigraph* ist ein Graph mit der Eigenschaft, dass mehrere Kanten zwischen zwei Knoten parallel existieren können. Um zwei Kanten voneinander zu unterscheiden, existiert die *labeled*-Eigenschaft, welche eindeutige Bezeichner bereitstellt. Die Kante selbst wird wie in der *directed*-Eigenschaft benannt, jedoch mit unterschiedlichem Index i als Bezeichner der Kante. Somit wären $e'_i = (v_a, v_b)$ und $e'_j = (v_a, v_b)$ zwei komplett verschiedene Kanten.

Diese vier Attribute helfen dabei, die Struktur im System zu erfassen und zu quantifizieren, sodass notwendige Zähloperationen ausgeführt werden können. Ein einfaches Beispiel eines solchen Graphen ist in Abbildung 5.3 zu finden. Wie genau die vier Eigenschaften für den Nachweis relevant werden, wird in den folgenden Kapiteln genauer erläutert. Für weitere und allgemeine Informationen zu diesen Eigenschaften von Graphen kann bei Interesse entsprechende Fachliteratur herangezogen werden [13] [31]. Wichtig ist vorerst, dass die oben genannten vier Eigenschaften durchaus noch ohne Semantik vorliegen. Es sind „nur“ Eigenschaften eines Graphen und bisher ist noch keinerlei Interpretation angewendet worden, was genau diese Eigenschaften bezüglich Softwaresystemen aussagen sollen oder können. Darum soll es im folgenden Kapitel 5.2.1 „Erstellung des Graphen“ gehen. Es wird erklärt, inwieweit Knoten im Graph die Komponenten darstellen, die Kanten bestimmte Beziehungen referenzieren, was genau daran *annotated* und *directed* bedeutet.

Bei der Entwicklung des Graphen spielten die zwei folgenden Fragen eine entscheidende Rolle:

1. Worin äußert sich Redundanz im System?

2. Wie beziffert man den Schaden durch die Redundanz?

Es stellt sich nach einiger Betrachtung heraus, dass dies zwei unterschiedliche Phasen im Nachweis bilden. Die erste Frage geht darauf ein, wie man die Redundanz im System herausfiltert und entsprechend im Graphen wiedergibt. Dabei ist es einfach, das Redundanz-Phänomen zu erklären, wie bereits in Kapitel 4 „Redundanz“ geschehen, allerdings ist es umso schwieriger,

konkrete Punkte im System dafür zu suchen, zumal verschiedene Redundanz-Arten auch verschiedene Äußerungen im System besitzen. Die Datenredundanz besitzt direkte Auswirkungen auf Komponenten und Daten im System, weswegen diese beiden Elemente die Hauptakteure im Graphen sein werden. Hat man einmal die strukturelle Äußerung von Redundanz im System festgelegt und festgestellt, ergibt sich durch die zweite Frage die Notwendigkeit einer Messbarkeit der Darstellung. Auch bei diesem Punkt sind viele verschiedene Möglichkeiten gegeben den Wert zu definieren: als imaginäre Einheit, als festen Wert, als relativen Wert oder auch als Werte-Verbund mit mehreren Faktoren. Durch die einigermaßen praxisorientierte Motivation dieser Arbeit wäre demzufolge ein fester Wert sinnvoll, welcher geeignet ist, in „hartem Cash“ wie Euro oder Dollar umgerechnet zu werden. Die einzelnen Metriken, welche am Ende wert-bestimmend wirken, sind im nächsten Kapitel 5.3 „II: Die Metriken“ genannt. Im aktuellen Kapitel soll es vorerst um den Graphen gehen, welcher als Grundlage für die Darstellung des Nachweises dienen soll, mit all seinen Eigenschaften und Methoden.

5.2.1 Erstellung des Graphen

Dieser Abschnitt verbindet nun die mathematische Definition eines *labeled, annotated, directed Multigraph* mit der in dieser Arbeit benutzen Definition. Dies ist notwendig, um die kausale Kette an Schlussfolgerungen zu erhalten und dabei die Korrektheit des Nachweises weiterhin zu stützen.

Es wird hierbei genau auf die vier oben genannten Punkte eingegangen, welche auch im Namen des Graphen enthalten sind: Labels, Annotationen, gerichtete Kanten und Parallelität. Das Ganze wird dabei in den vier Schritten stattfinden:

1. Ausleihe des mathematischen Modells
2. Anwendung eigener Semantiken
3. Anreicherung des Modells
4. Entwurf eigener Algorithmen

Wenn diese vier Schritte abgeschlossen sind, ist der eigentliche Graph aus dieser Arbeit umfangreich aus der mathematischen Definition hergeleitet. Es sei hierbei angemerkt, dass die beiden folgenden Abschnitte 5.2.2 „Eigenschaften“ und 5.2.3 „Evolution“ näher auf die einzelnen Aspekte des Graphen eingehen und das derzeitige Kapitel nur für die Herleitung dienen soll.

Das in Abbildung 5.3 gezeigte Beispiel eines Graphen soll am Ende ein Vorbild für den Graph dieser Arbeit darstellen. Die dazu nötigen mathematischen Eigenschaften des *labeled, annotated, directed* und Multigraph müssen daher sinnvoll in die entsprechenden Bereiche dieser Arbeit abgebildet werden. Mit der „Ausleihe“ des Modells, das heißt des Graphen, kann die eigene Arbeit darauf aufbauen. Damit sind die Grundsteine an Definitionen gelegt und es kann damit begonnen werden, die einzelnen Elemente des Graphen mit eigenen Semantiken zu versehen. Zuerst müssen alle strukturellen Elemente des Graphen definiert werden, bevor man zu den Eigenschaften übergehen kann. Deswegen wird ein Graph in dieser Arbeit wie folgt definiert, wobei genauere Angaben zu den Elementen im nächsten Abschnitt erläutert werden.

1. Der Graph $G(V, E)$ wird umgesetzt in $G(\{\mathbb{C}, \mathbb{D}\}, \mathbb{X})$ als Menge (das System) von Komponenten und Datenbeständen in Verbindung mit Kanten aus der Menge $\mathbb{X}$ (diese wird später spezifiziert).

2. Die Knoten des Graphen stellen Komponenten $c \in \mathbb{C}$ und Datenbestände $d \in \mathbb{D}$ dar.

3. Die Kanten des Graphen werden Beziehungen aus der Menge $\mathbb{X}$ sein, welche die Interaktion von Komponenten und Datenbestände beschreiben.

Die vier genannten Eigenschaften eines *labeled, annotated, directed Multigraph* werden nun dazu wie folgt umgesetzt.

1. *labeled* wird zu Labeln von Elementen des Systems. Sowohl Komponenten als auch Datenbestände sollten ein numerisches Label besitzen, welches sie eindeutig und dauerhaft identifiziert. Damit ist sichergestellt, dass alle Elemente des Systems sicher angesprochen werden können. Dadurch ergibt sich aus dem mathematisch definierten Graphen das System $\mathbb{E}(\{\mathbb{C}, \mathbb{D}\}, \mathbb{X})$ mit $c_i \in \mathbb{C}$, $d_j \in \mathbb{D}$ und $x_k \in \mathbb{X}$ (die Kanten) mit jeweils $i, j, k \in \mathbb{N}^+$.

2. *annotated* wird auf Eigenschaften des Systems bezogen, was bedeutet, dass bestimmte Kernpunkte aus der Darstellung als Faktoren eingehen, welche eine Annotation (also eine Eigenschaft) im Graph bekommen. Dabei wird die gängige Notation mit $A(c_i)$, beziehungsweise A_{c_i}, benutzt, wobei das A eine erforderliche Eigenschaft einer Komponente c_i darstellt. Für Datenbestände d_i oder Beziehungen x_i funktioniert das analog.

3. *directed* wird der Graph dadurch, dass die mathematisch definierten gerichteten Kanten mit $e' = \langle v_i, v_j \rangle$ in ein entsprechendes Äquivalent umgewandelt werden. Damit ist eine Kante in dem neuen Graphen definiert mit $x = \langle y_i, y_j \rangle$, wobei das $x \in \mathbb{X}$ eine der neuen Kantenarten darstellt und $y \in \{\mathbb{C} \cap \mathbb{D}\}$ gilt.

4. *Multigraph* wird dadurch umgesetzt, dass parallele Kanten existieren können. Sollte dies vorkommen, ist durch die *labeled*-Eigenschaft dafür gesorgt, dass die einzelnen Kanten noch unterscheidbar bleiben. In dem Graphen dieser Arbeit sind parallele Kanten nur zwischen Redundanzen möglich.

Damit sind die Eigenschaften definiert und die neue Semantik hinter den Elementen des Graphen festgelegt. Der dritte Schritt, die Anreicherung der Graphen-Definition, wird dadurch realisiert, dass der normale *labeled, annotated, directed Multigraph* „mehr“ enthält als jeweils nur einen Fall pro Definition. Als Beispiel wird eine Anreicherung der *annotated*-Eigenschaft dadurch erreicht, dass diese auf Knoten und Kanten ausgeweitet wird und es mehrere Annotationen pro Element im Graphen geben kann. Ebenso ist es Inhalt der Anreicherung, dass neue Eigenschaften oder Invarianten des Graphen erreicht werden. So ist eine dieser neuen Eigenschaften, dass eine Komponente niemals eine Senke im „Fluss“ der gerichteten Kanten im Graphen sein kann. Warum dies so ist, wird in den folgenden Kapiteln dargestellt, wenn die einzelnen Elemente des Graphen genauer untersucht werden.

Der wichtigste Punkt ist der vierte Schritt, der Entwurf eigener Algorithmen. Zu einem System die Metriken zu entwickeln bedeutet, auf dem hier entstehenden Graphen neue Zähloperationen zu entwickeln, sofern noch nicht vorhanden. Die neue Problemstellung, beziehungsweise die Hypothese, gibt einen solchen neuen Rahmen vor und erfordert neue Algorithmen auf Basis des Graphen mit seinen Eigenschaften. Es gibt unzählige mathematische Algorithmen, welche auf Graphen angewendet werden können, jedoch erfordert es die Zielstellung, dass neue Zähloperationen auf Komponenten, Datenbanken und deren Beziehung zueinander ausgeführt werden. Es wurde dabei in Anlehnung an mathematische Eigenschaften des Graphen gearbeitet, wie zum Beispiel der „maximale Schnitt“ eines Graphen. Aber abseits dieser Eigenschaften wurden keine Algorithmen, wie zum Beispiel „kürzeste Wege“, verbaut oder anderweitig genutzt. Wie genau die

neuen Algorithmen, das heißt die späteren Metriken, aussehen und was diese bewirken, wird im Kapitel 5.3 „II: Die Metriken“ erläutert. Die beiden folgenden Abschnitte klären die neuen Eigenschaften und Verhaltensweisen des entstandenen Graphen und zeigen dazu auch Abbildungen und Beispiele auf. Da ab hier der „wirkliche“ Nachweis beginnt, werden ab nun auch notwendige Annahmen entsprechend gekennzeichnet.

5.2.2 Eigenschaften

Die Kernfähigkeit des entstandenen Graphen ist die Möglichkeit, für die Quantifizierung notwendige Werte abzubilden. Die dafür notwendigen nummerischen Werte sind in Attributen des Graphen erfasst. Es sind natürlich noch viele Werte des Systems nicht auf den Graphen abgebildet, jedoch ist es für die Kausalität des Nachweises sinnvoller, nur die wichtigsten Eigenschaften abzubilden. Diese Systemeigenschaften werden durch das *annotated*-Attribut des Graphen abgebildet, indem Knoten und Kanten selbst verschiedene Werte beinhalten können. In Dimitropoulos *et al.* [14] wurde bereits ein annotierter Graph erfolgreich für Untersuchungen von Verhalten und Komplexität eines Netzwerks benutzt.

Allerdings spielt es nicht nur eine Rolle, welche Eigenschaft mit in den Graphen aufgenommen wird, sondern auch, inwieweit man eine Eigenschaft sinnvoll einbauen kann. Ein für den Graphen dieser Arbeit aufgetretenes Beispiel wäre hierfür die Eigenschaft der „Nutzung“ eines Elements. Auf den ersten Blick kann man sich durchaus vorstellen, dass dies eine relevante Eigenschaft darstellt, welche diverse Aussagen über Fehleranfälligkeit, Stabilität und andere Eigenschaften gibt. Die eigentliche Frage dahinter ist allerdings, wie man diese abstrakte Eigenschaft der „Nutzung“ im Graphen umsetzt, immerhin ist es ein für den Nachweis erforderlicher Wert. Die Möglichkeiten dafür, eine Eigenschaft umzusetzen, sind dabei nicht gering: man kann die „Nutzung“ als Kante, als Knoten, als Eigenschaft eines der beiden Elemente definieren, oder aber die „Nutzung“ aufsplitten in „Zugriff“, „Speicherung“ oder auch „Berechnung“. Viele Möglichkeiten stehen offen, es gilt jedoch die geeignete zu finden, welche den Nachweis hinsichtlich der Hypothese stützen kann.

Wie in Kapitel 3.2 „Softwaresysteme als Graphen“ bereits geschrieben, wird für den Nachweis der Graph unterteilt in *Elemente* und deren *Beziehungen*. Diesen beiden Teilen ist eine notwendige Formalisierung zuzuordnen, damit daraus ein geeignetes Gerüst für den Nachweis entstehen kann. Aber entgegen der eventuell auftretenden Vermutung, an dem formalen Graphen diverse mathematische Algorithmen auszuführen, nutzt man doch hauptsächlich „nur“ Zähloperationen auf den Elementen, um in der Kette der logischen Schlussfolgerungen voranzukommen. Es wäre durchaus möglich, Algorithmen wie „kürzeste Wege“, „maximaler Fluss“ und weitere auszuführen, da die dafür notwendigen Eigenschaften in dem *labeled, annotated, directed Multigraphen* vorhanden sind, nur sind diese leider nicht wirklich geeignet, um am Ende auf den Wert durch Redundanz zu schließen. Das bedeutet im Umkehrschluss aber auch, dass solche Algorithmen durchaus nützlich sein können, wenn es um den Wert anderer Strukturen im System geht. Da dies in dieser Arbeit nicht der Fall ist, wird davon abgesehen, die vollen Möglichkeiten eines formalen strukturierten Graphen zu nutzen.

Das `init`-System Die Knoten in dem in dieser Arbeit erstellen Graphen sind zweigeteilter Natur: einerseits werden Knoten benutzt, welche für *Komponenten* im System stehen, andererseits gibt es eine zweite Knoten-Art für sogenannte *Datenbestände* im System. Dies ist in Abbildung 5.4 veranschaulicht: man kann die Komponenten (die viereckigen Knoten) sowie die Datenbestände (die runden Objekte) erkennen.

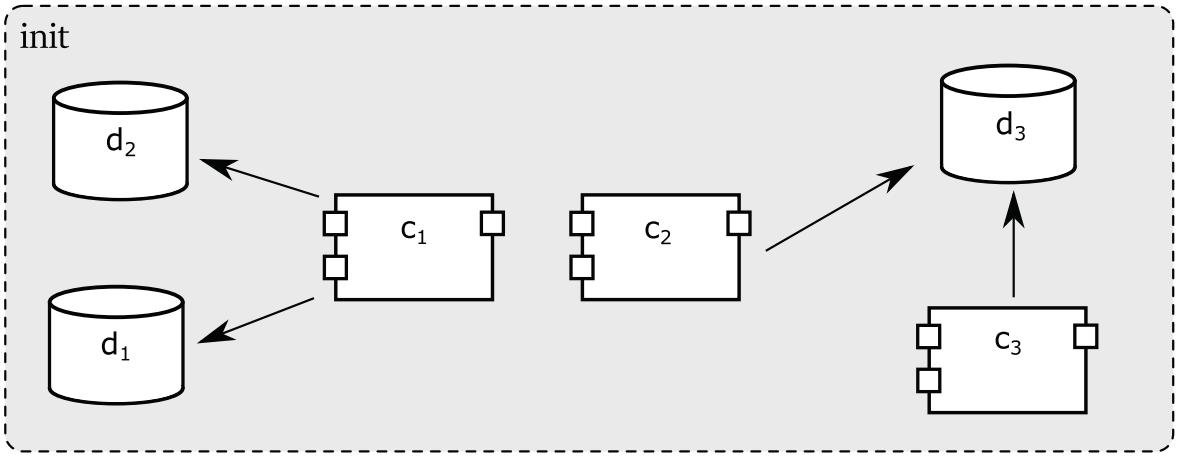


Abbildung 5.4: Die Knoten und Kanten des *labeled, annotated, directed Multigraphen*

Wie man der Abbildung entnehmen kann, sind die Knoten entsprechend mit c und d gelabelt. Ebenso kann man erkennen, dass bereits erste Kanten aufgezeigt werden. Es gibt natürlich noch mehr Elemente in einem Softwaresystem, welche eine eigene Knoten-Art verdient hätten. Aber auch hierbei wurde nur das einbezogen, was für eine Datenredundanz relevant ist, und dabei sind hauptsächlich Komponenten und Datenbestände beteiligt. Dadurch ergibt sich die erste Annahme, welche im Zuge der logischen Nachvollziehbarkeit getroffen werden muss.

Annahme 1
Für die Darstellung von Datenredundanz ist es hinreichend, wenn Komponenten und Datenbestände als Knoten beschrieben werden.

Diese Art der Vereinfachung des Graphen wird allerdings noch des Öfteren auftauchen. Das einfache System, wie es in der Abbildung aufgezeigt wird, besteht in seiner Grundphase nur aus je drei Komponenten (c) und Datenbeständen (d). Das *init* in der oberen linken Ecke betitelt einfach vorerst nur die Grundversion, wie sie gerade vorliegt. Formal gesehen wäre das System $\mathbb{E}$ im Zustand

$$\mathbb{E}_{\text{init}} = \mathbb{E}_0^+ = \{\mathbb{C}_0^+ \cap \mathbb{D}_0^+ \cap \mathbb{X}_0^+\} \text{ mit } c_1, c_2, c_3 \in \mathbb{C}_0^+ \text{ und } d_1, d_2, d_3 \in \mathbb{D}_0^+$$

in der allerersten Erweiterung (*init*), welche demzufolge keine Erweiterung im eigentlichen Sinne darstellt sondern einfach der Anfang des Systems ist und damit mit 0 beziffert wird. Das System besteht damit nur aus einer Erweiterung und somit ist die Erweiterung gleich dem System an sich, was sich jedoch mit zukünftigen Erweiterungen ändert. Um eine Erweiterung im Graphen zu markieren, ist die grau gefärbte Box hinter den dazugehörigen Elementen vorhanden. Dies ist jedoch nur eine Darstellungsform der Erweiterung, man könnte auch die einzelnen Elemente des Systems entsprechend separat einfärben, was genau dann zu bevorzugen ist, wenn der Graph nicht mehr mit rechteckigen Boxen separierbar wird.

Die Kanten des Graphen, welche in Abbildung 5.4 zu sehen sind, stellen im weiteren Sinne die Benutzung dar. Es gibt verschiedene Arten von Kanten in dem entstandenen Graphen, die Basis-Variante ist der spitze Pfeil mit ausgefülltem Pfeilkopf. Dieser bildet die Beziehung zwischen einer Komponente und einem Datenbestand ab. Dabei bedeutet dieser, dass mindestens eine Funktionalität der Komponente auf den Datenbestand zugreift. Diese Ansicht wird später

verfeinert, dies soll vorerst nur den Grundgedanken aufzeigen. Darüber hinaus gibt es bislang keine Beziehungen zwischen je zwei Komponenten oder Datenbeständen untereinander, da dies in dem Nachweis entweder nicht benötigt oder erst in einem späteren Schritt abgebildet wird.

Der Grund, weshalb in diesem Nachweis vom Wert durch Softwarearchitektur von einem imaginären und idealen Neu-System ausgegangen wird, ist bedingt durch die Natur des Nachweises, der am Ende feststellen sollte, dass ein ideales System durch Veränderungen an dessen Struktur (*unmanaged* Datenredundanz) ein Wertverlust vorliegt. Um die Ausmaße des Verlustes zu ermitteln, ist es demzufolge nötig, den Idealwert vorher zu kennen. Es ist durchaus nachvollziehbar, dass diese idealen *unmanaged* Redundanz-freien Systeme so gut wie nie in der realen Praxis vorkommen, aber das ist letztlich auch nicht die Kernaussage des Nachweises, wie in 1.2 „Motivation für den Nachweis von Werten der Architektur“ geschildert.

Annahme 2
Das `init`-System ist immer frei von unmanaged Redundancy.

Anhand dieser Basis-Struktur von Knoten und Kanten kann ein System durchaus sinnvoll abgebildet werden. Es ist verständlich, dass hierbei einige wichtige Eigenschaften des Systems wegfallen und nicht gut darstellbar sind, was aber für den eigentlichen Nachweis keine Rolle spielt, da diese Eigenschaften nicht benötigt werden. Allein durch das Abbilden von Komponenten (zum Beispiel Klassen, Methoden, Module oder Frameworks), Datenbeständen (zum Beispiel Datenbanksysteme, Tabellen, Speicherdateien oder auch hart-codierte Daten) sowie deren Beziehung zueinander, lässt sich die Datenredundanz sehr gut erklären und formal beschreiben, was wohl die wichtigste Eigenschaft dieser Methode ist. Im Folgenden werden die einzelnen, dem Graphen zugehörigen Annotationen und Eigenschaften aufgezeigt, wobei zu beachten ist, dass auf die eigentlichen Metriken dazu, wie genau sich diverse Werte berechnen, erst im nachfolgenden Kapitel detailreich eingegangen wird. Hier soll vorerst nur erläutert werden, welche Eigenschaften der entstandene Graph umsetzt.

Die Nutzrate von Komponente zu Datenbestand Um die Beziehung zwischen Komponente und Datenbestand genauer zu definieren, stellt der Graph eine Annotation für diese Pfeile bereit, die sogenannten *CRUD*-Werte. Man kennt diese Werte vielleicht bereits aus der Datenbanktechnologie, welche für die vier folgenden Informations-bezogenen Aktionen stehen und die Handhabung derer genauer spezifizieren:

C steht für *Create* („erstellen“) und benennt die Aktion bei Datenbanken, eine neue Information zu erstellen und abzuspeichern.

R ist der Wert für ein *Read* („lesen“) und meint damit alle Aktionen, welche ganze oder Teile von Informationen auslesen.

U ist der *Update*-Wert und beziffert alle Aktionen auf dem Datenbestand, welche Informationen verändern, das heißt insbesondere Teile dieser Information aktualisieren oder abzuändern.

D ist der Wert für *Delete*-Aktionen („löschen“) bei Informationen. Hierdurch wird eine Information aus dem Datenbestand entfernt.

Zu Beginn des Nachweises stand die Option, statt dem *CRUD* einfachere Klassifikationen wie *R/W* (*Read- and Write*) für Lese- und Schreib-Zugriffe zu nutzen. Jedoch ist die *CRUD*-Klassifikation besser für die Darstellung und den Graphen geeignet, da zum Beispiel ein *Update*

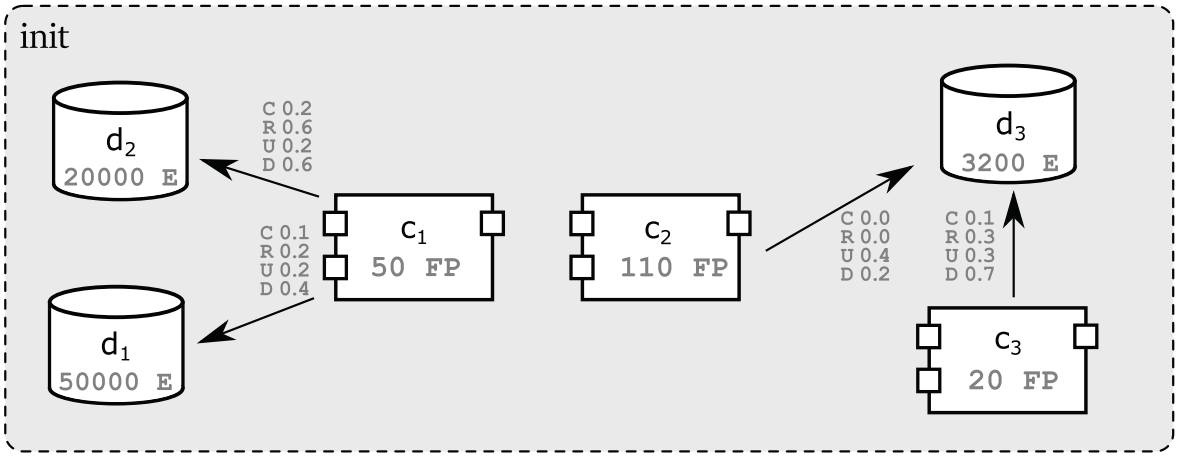


Abbildung 5.5: Die Annotationen der Knoten und Kanten

der Daten andere Auswirkungen (bezüglich der *unmanaged Datenredundanz*) besitzen kann als ein *Delete*, was in der *R/W*-Klassifikation als eine Kategorie angesehen und damit vereinheitlicht werden würde. Um nun diese *CRUD*-Werte sinnvoll in den Graphen als Annotation einzubauen, musste definiert werden, inwieweit diese die Benutzung zwischen Komponente und Datenbestand regeln. Diese *CRUD*-Werte sind demnach ein vierdimensionaler Vektor $\overrightarrow{CRUD}$ als Eigenschaft des „Benutzen"-Pfeils und basiert auf den einzelnen Aktionen, inwieweit (relativ gesehen) die verschiedenen Aktionen einer Komponente in den Kategorien der *CRUD*-Klassifizierung mit dem Datenbestand arbeiten.

Annahme 3
Das Verhalten der Benutzung von Komponenten zu Datenbeständen ist für Datenredundanz relevant und kann durch das CRUD-System hinreichend beschrieben werden.

Die genauere Metrik, inwieweit man die spezifischen *CRUD*-Werte berechnet und wozu diese später beitragen werden, wird im folgenden Kapitel 5.3 „II: Die Metriken" beschrieben. Im Graphen äußert sich dieser Vektor als einfache Pfeil-Annotation, wie in dem nun mit Annotationen versehenen Beispiel-System aus Abbildung 5.5. Dabei trägt jede Komponenten-Datenbestand-Beziehung einen eigenen Vektor für die *CRUD*-Werte.

Die Knoten-Annotationen Wie ebenfalls in Abbildung 5.5 zu erkennen ist, besitzen auch die Knoten verschiedene Annotationen. Die Komponenten des Systems, also die viereckigen Knoten c_1, c_2, c_3, besitzen ihrerseits die Eigenschaft der *Function Points* (*FP*), welche bereits einmal angesprochen wurde. Diese *FP*-Eigenschaft ist dafür gedacht, den inhaltlichen Umfang der Komponenten darzulegen, unabhängig ihrer Implementierung sondern definiert durch die Anforderungen an diese sowie deren Funktionalitäten. Eine große Komponente mit viel *#LOC* kann damit durchaus auch nur wenige *FP* besitzen, und umgekehrt. Diese Entscheidung ist dahingehend wichtig, dass durch *FP* auch der Aufwand und das Investment der einzelnen Komponenten festgestellt werden kann und damit als Vergleich bei Wertverlust dienen kann, wo andere Metriken wie *#LOC* nicht zielführend anwendbar sind. Analog zu *FP* kann auch eine andere, inhaltsbasierte Metrik benutzt werden, wie zum Beispiel die bereits bekannten *Use Case Points* oder auch *Object Points*. Wichtig ist nur, dass die inhaltliche Größe einer Komponente

zählt und die gewählte Metrik konsistent im Graphen und dem Nachweis benutzt wird. Einen guten Überblick über die einzelnen genannten Größen und deren eigentliche Berechnung ist in Fenton *et al.* [16] gegeben. Es ist für den Zweck dieser Arbeit wichtig, dass die *FP* als Aufwand verstanden werden können und damit direkt in Verbindung mit *TtM* sowie *DevC* stehen. In der Statistik wird angegeben, dass ein einzelner Entwickler je nach Kenntnisstand und Umfeld ungefähr zwei Tage für einen *FP* benötigt [16], womit sich automatisch eine *TtM* ergibt und indirekt als „Bezahlung“ mindestens der Lohn für den Entwickler in die *DevC* eingeht.

Die Datenbestands-Knoten besitzen in dem Graphen ebenfalls eine eigene Annotation. Da die eingeführte Größe der *Function Points* bei Datenbeständen keinen wirklichen Sinn ergibt, wurde eine andere Größe für Datenbestände genutzt, die der *Entities* (Einheit). Im Gegensatz zu den Komponenten eines Systems, wo die Anzahl an Codezeilen stark von der Implementierung abhängig ist, ist bei Datenbeständen eine Information auch meist immer ein Eintrag. Dieser kann bei Datenbanken durchaus auf mehrere Tabellen oder andere Strukturen verteilt sein, insgesamt kann man aber mit der Zählung der *Entities* die Größe der Information recht zuverlässig abschätzen. Ein Datenbestand mit zum Beispiel tausenden *Entities* ist immer mit mehr Informationen belegt als ein Datenbestand mit ein paar Hundert *Entities*. Das gilt weitestgehend für alle Arten der Speicherung von Informationen: eine Datenbank kann die *Entities* zeilenweise mit Referenzen abspeichern, aber auch hart-codierte Informationen können als einzelne *Entities* angerechnet werden, zum Beispiel alle in einer `HashMap` gespeicherten Wertepaare. Um die *Entities* im Graphen darzustellen, besitzt jeder Knoten vom Typ „Datenbestand“ ein Attribut *E* mit einer zugeordneten positiven, natürlichen Zahl. Dabei sei hier erwähnt, dass die *Entities* im Sinne der Datenbanktechnologie nicht genau den *Entities* in Bezug auf die Datenbestände in dieser Arbeit entsprechen. Obwohl beides die gleiche Bedeutung haben kann, ist es dennoch möglich, dass Datenbestände mit *Entities* belegt sind, welche keine Datenbanken sind. Deswegen wird in dieser Arbeit eine *Entity* auch eher als eine Informationseinheit angesehen, weniger als eine Zeile in einer Liste oder Tabelle.

Annahme 4
Der inhaltliche Umfang ist für Datenredundanz relevant und kann durch die Annotationen an Knoten für Komponenten und Datenbestände (Function Points und Entities) hinreichend beschrieben werden.

Die Redundanz-Beziehung Eine letzte große Eigenschaft des Graphen gilt es noch genauer zu erläutern, die Beziehung der Redundanz, wie in Abbildung 5.6 durch die roten und grünen Kanten aufgezeigt. Man kann nebenbei erkennen, dass das System um die beiden Erweiterungen *ext1* und *ext2* bereichert wurde, was erst im folgenden Abschnitt 5.2.3 „Evolution“ thematisiert wird. Die Kanten, welche die Redundanz darstellen, sind vom zweiten Kanten-Typ im Graphen und werden mit *Rd* benannt, abgeleitet vom Wort „Redundanz“. Der Wert des Attributes der Kante gibt an, inwieweit ein Datenbestand bestimmte Werte von einem anderen kopiert hat. Dabei sei angemerkt, dass hierbei die Einzelteile einer Information eine Rolle spielen, sodass auch nur teilweise kopierte Informationen (*Entities*) bereits zu einer Erhöhung des Redundanz-Attributes führen. Allgemein kommt es bei dem Wert der Redundanz nicht wirklich auf den Namen der kopierten Werte an, sondern vielmehr um den Inhalt, sodass auch kopierte und anderweitig benannte und genutzte Werte in einer Erhöhung des Redundanz-Attributes resultieren. Ausführungen dazu, inwieweit der Wert des Attributs gebildet wird, wird ebenfalls wie die anderen Einheiten im Kapitel 5.3 „II: Die Metriken“ aufgezeigt.

Wie man erkennen kann, gibt es innerhalb der Kanten vom Typ „Redundanz“ verschiedene

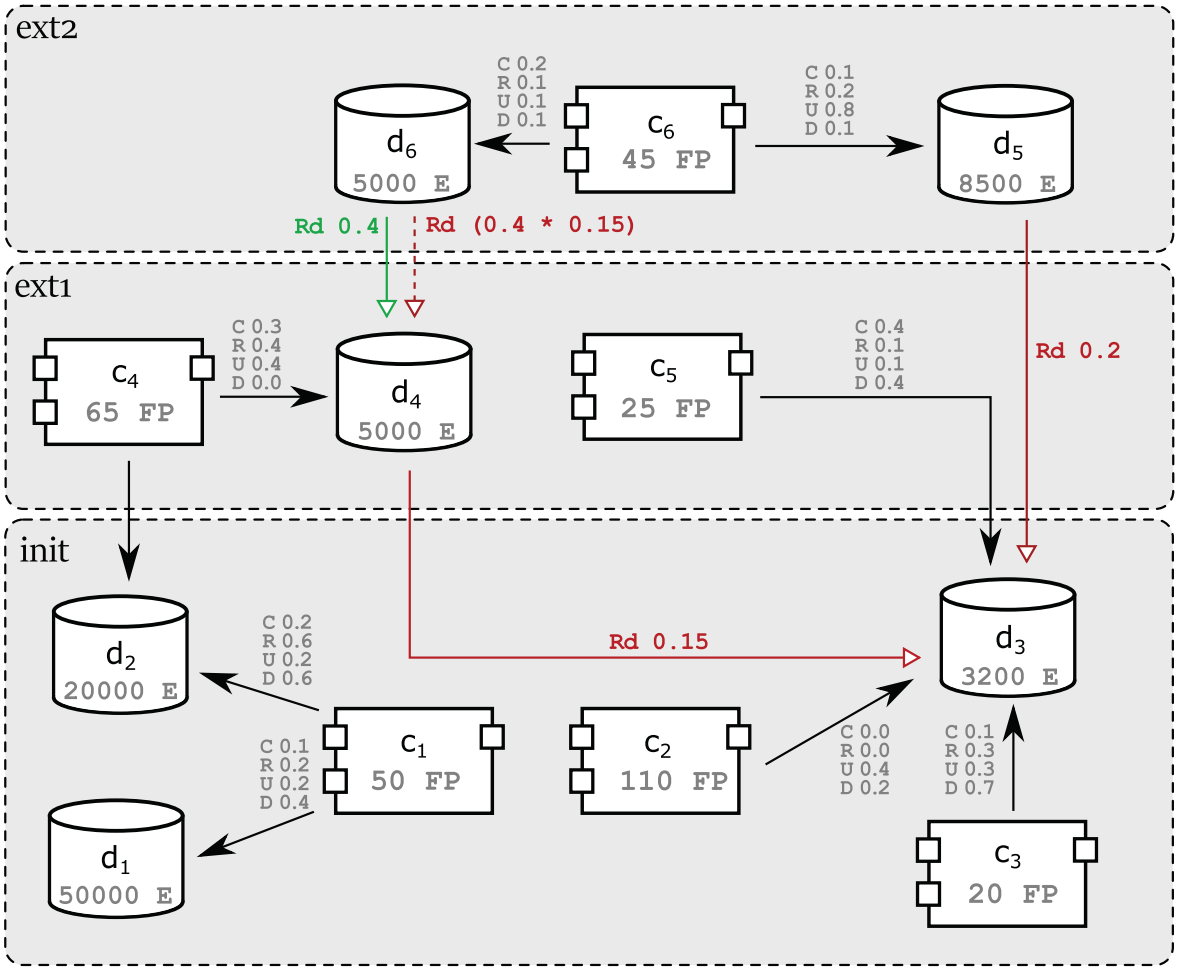


Abbildung 5.6: Die Redundanz-Kanten und Erweiterungen im Graphen

Ausprägungen. Die gemeine rote Kante ist die „Standard-Redundanz“, also das exakte Kopieren (per *unmanaged* Datenredundanz) von Werten aus erster Hand, formal betitelt als $\overline{Rd}_{e'_i} = n$ mit n als der Attributs-Wert, welcher im Graphen neben der Kante eingezeichnet ist. Die Standard-Redundanz ist als durchgehender roter Pfeil im Graphen markiert. Dabei ist zu beachten, dass im Graphen der Strich über dem *Rd* nicht vorhanden ist, da eine durchgezogene rote Kante *immer* eine *unmanaged Redundancy* vom Typ „Standard“ darstellt. Ein Strich über dem Wert im Graphen würde damit eine redundante Information in der Formalisierung dieser Arbeit bedeuten, weswegen dieser Querstrich nur in Formeln zur Anwendung kommt.

Der nächste Untertyp von Redundanz ist die „synchronisierte Redundanz“ (also *managed* Datenredundanz), im Graphen als grüner Pfeil dargestellt. Diese Redundanz kopiert zwar ihrerseits ebenfalls Werte von anderen Datenbeständen, jedoch wird hierbei sichergestellt, dass die kopierten Werte inhaltlich, zeitlich und ausfallsicher immer synchronisiert bleiben, sodass diese *managed* Redundanz keine negativen Folgen bezüglich des Wertes des Systems besitzt. Natürlich kann eine synchronisierte Redundanz einen positiven Einfluss auf das System haben, zum Beispiel zur Steigerung der Performance, was jedoch nicht die Thematik in dieser Arbeit trifft und damit näher untersucht wird. Eine synchronisierte Redundanz wird hierbei formal

dargestellt als $Rd_{e'_i} = n$. Der Wert n, also der Wert des Attributs an der entsprechenden Kante im Graphen, ist dennoch wichtig, da dies für den letzten Typ von Redundanz-Kante benötigt wird.

Definition 1
Eine Synchronisierung eines Datenbestandes ist genau dann hinreichend, wenn sichergestellt ist, dass dieser Datenbestand inhaltlich und zeitlich konsistent und arbeitsfähig gehalten wird, was je nach Anforderung an den Datenbestand unterschiedlich ausfallen kann.

Dieser letzte Typ von Redundanz ist die „vererbte Redundanz“ (ein Untertyp der *unmanaged* Datenredundanz), im Graphen dargestellt als gestrichelter roter Pfeil und ebenfalls mit einem Redundanz-Attribut versehen. Formal wird eine solche Kante in dieser Arbeit mit $\widehat{Rd}_{e'_i} = n$ dargestellt. Eine solche vererbte Redundanz existiert genau dann, wenn eine synchronisierte Redundanz verschiedene Werte aus einem Datenbestand synchronisiert kopiert, dieser selbst allerdings von woanders mit einer Standard-Redundanz Werte kopiert. Dafür ist der Wert von der synchronisierten Redundanz notwendig, da die entsprechende Höhe der vorherigen Standard-Redundanz nun weiter propagiert wird. Um die Höhe der vererbten Redundanz zu bestimmen, ist die Höhe der synchronisierten Redundanz sowie der darunter liegenden *unmanaged Redundancy*'s via Standard-Redundanz notwendig. Durch die Multiplikation beider Werte entlang des Pfades ergibt sich dann die Höhe der vererbten Redundanz. Sollte es sich um mehrere Datenbestände handeln, aus denen kopiert wird, werden dabei alle Beteiligten beachtet. Dadurch wird die eigentliche Höhe der synchronisierten Redundanz nie direkt in Metriken einbezogen, sondern nur über deren Einfluss in die vererbten Redundanzen. Weitere Erläuterungen zu der Metrik über die Höhe der Redundanz sind im Kapitel 5.3 „II: Die Metriken“ enthalten. Wichtig ist vorerst, dass eine vererbte Redundanz damit nur den Teil einer anderen Standard-Redundanz darstellt, welche die entsprechend zugehörige synchronisierte Redundanz ungewollt mitnimmt. In Rahmen dieser Arbeit wurde entschieden, dass eine vererbte Redundanz nur über eine Generation nach der letzten Standard-Redundanz möglich ist, was bedeutet, dass eine erneute vererbte Redundanz von einer bereits bestehenden vererbten Redundanz nicht möglich ist, genauso wie eine Verkettung von anderen Redundanzen sich nicht auswirkt. Dies ist in Abbildung 5.7 noch einmal genauer grafisch aufgeführt. Man kann in dieser Abbildung erkennen, dass der Datenbestand d_1 normalerweise drei vererbte Redundanzen besitzt. Jedoch kommt nur die vererbte Redundanz B infrage, da dies die Einzige ist, welche über nur eine Generation vererbt ist. Die Redundanz A wird wegen der Verkettung von zwei synchronisierten Redundanzen ausgeschlossen ($[a]$), die Redundanz C wegen der Verkettung von zwei Standard-Redundanzen ($[b]$), wobei eine vererbte Redundanz nur aus synchronisierten Redundanzen entstehen kann. Dass diese verketten Redundanz-Phänomene in der Praxis vorkommen können, ist bekannt, jedoch ist es ein sehr hoher Aufwand, diese im Graphen zu modellieren. Des weiteren wird der Schaden am System theoretisch nur minimal erhöht und ist damit vernachlässigbar. Man beachte, dass in Abbildung 5.7 die „vererbte Redundanz“-Kanten, anders als im normalen Graphen, neue Verbindungen zwischen Datenbeständen schaffen, welche vorher nicht vorhanden waren (zum Beispiel die Kante $[b]$). Im normalen Graphen wird die vererbte Redundanz *immer* parallel zur synchronisierten Redundanz vorkommen (wie in Abbildung 5.6 dargestellt), was an der dazu definierten Metrik liegt. Weitere Ausführungen dazu folgen im Kapitel 5.3 „II: Die Metriken“.

Mit diesen verschiedenen Arten der Redundanz ergeben sich neue Annahmen, welche für die Darstellung des Nachweises und dessen spätere Messbarkeit eine erhebliche Rolle spielen.

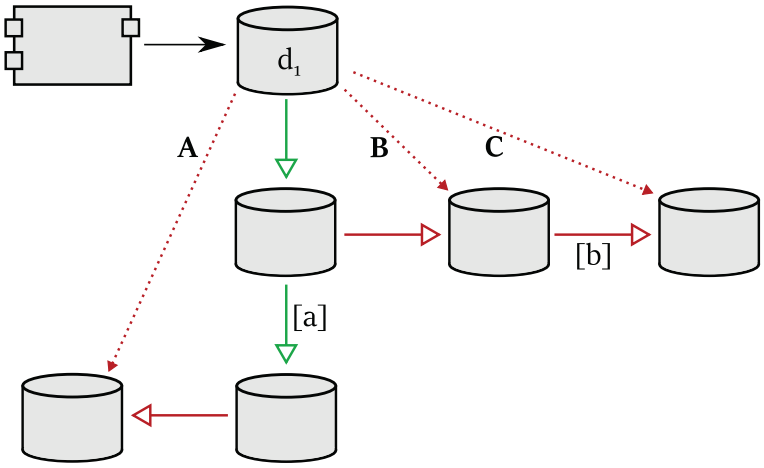


Abbildung 5.7: Die Funktionsweise der vererbten Redundanz: nur *B* wird beachtet

Annahme 5
Die Datenredundanz im System ist mit der Standard-Redundanz, der vererbten Redundanz und der synchronisierten Redundanz hinreichend beschreibbar.

Annahme 6
Eine vererbte Redundanz, welche aus einer verketteten Redundanz entsteht, verursacht nur noch vernachlässigbaren Schaden. Ebenso sind vererbte Redundanzen aus bereits vererbten Redundanzen vernachlässigbar.

Annahme 7
Eine Standard- und vererbte Redundanz ist, unabhängig von der entsprechenden Höhe, in jedem Falle schädlich für das System und eine synchronisierte Redundanz verhält sich neutral.

Die Notwendigkeit einen Nachweis für den Wertverfall durch Redundanz trotz der Annahme, dass eine *unmanaged Redundancy* immer negative Folgen auf den Wert des Systems ausübt, ist in den folgenden Punkten begründet:

1. Eine *unmanaged Redundancy* ist aus empirisch ermittelten Szenarien immer schädlich für das entsprechende System gewesen. Es gibt kaum Fälle, dass diverse Shortcuts (durch unkontrollierte Redundanz von Daten) beim Entwickeln von Systemen später in dessen Evolution Vorteile verschafft. Ein eingebauter Shortcut sorgt anfangs für kürzere *TtM* und *DevC*. Jedoch ist die Absicht dieses Nachweises eher in Bezug auf die Evolution zu sehen, inwieweit sich die durch den Shortcut eingebaute *Technical Debt* später auf das System auswirkt. Dabei ist der Einbau von *Technical Debt* bereits schon immer für zukünftige negative Folgen bekannt, jedoch steht bei vielen Dingen, wie hier bei der *unmanaged Redundancy*, der formale Nachweis noch aus.

2. Die möglichen negativen Folgen von *unmanaged Redundancy* sind durchaus bekannt. Für jeden der drei Bereiche *Adaptive*, *Corrective* und *Preventive Maintenance* ist aus real

existierenden Projekten bekannt, was passieren kann, wenn man diese entsprechend vernachlässigt, was auch bereits in Kapitel 2.1.2 „Die Kosten eines Systems“ behandelt wurde. Mit diesen negativen Folgen kann ein „Strafmaß“ definiert werden, welches dann anhand der Eigenschaften des Systems gewichtet wird. Daraus entstanden die in dieser Arbeit vorgestellten Graph- und Wert-Metriken. Dies stellt eine geeignete Methode dar, mit der man davon ausgehen kann, dass die Voraussetzung zutrifft. Es soll hierbei jedoch nicht ausgeschlossen werden, dass andere Methoden möglich wären.

3. Bei der Formalisierung des Verhaltens der *unmanaged Redundancy* ist es möglich, die Veränderungen im System durch die Redundanz zu simulieren. Die Frage, inwieweit sich die Erhöhung der *unmanaged Redundancy* im System auswirkt, ist bislang immer noch offen. Dabei sind die Fragen nach der Monotonie, dem Verhalten und dem Verhältnis in Bezug der einzelnen Werte noch ungeklärt. Durch die Formalisierung besteht anschließend die Möglichkeit, bestimmte Kenngrößen des Modells zu variieren, um deren Auswirkungen zu beobachten.

4. Der ganze hier beschriebene Nachweis ist entsprechend variabel gestaltet, sodass in Zukunft bei Bedarf verschiedene Änderungen durchgeführt werden können, ohne dass der gesamte Nachweis ungültig ist. Zum Beispiel wird sich der Verlauf einer logarithmischen Kurve durch einen zusätzlichen Faktor nicht in seinem Wesen verändern, sondern vielmehr die Höhe im Wertebereich, was aber die Aussage über das Verhalten nicht negiert.

5. Neben der Frage nach der Schädlichkeit von *unmanaged* Datenredundanz sind andere Fakten ebenso interessant, welche nicht zwangsweise von dieser Annahme abhängen, jedoch ebenso in dieser Arbeit ergründet werden, wie den Entwurf neuer Metriken für Architektur, die Formalisierung eines Software-Graphen, die Vorgehensweise und Methodik bei Nachweisen von Architektur und anderes. All das ist bereits ein großer Schritt in die richtige Richtung.

6. Am Ende steht natürlich auch die Frage nach der Schädlichkeit der *unmanaged Redundancy* im System. Obwohl dies durch die Annahme bereits vorausgesetzt wird, ist es bislang nicht genügend ergründet, warum diese Annahme zutrifft. Das bedeutet, auch wenn man mit dieser Annahme den Nachweis und die darauf aufbauende Quantifizierung erbringt, können dadurch ebenfalls die Gründe für die Annahme selbst herausgefunden werden.

Damit sind nun alle Elemente im Graphen ausreichend definiert, um später in der Messbarkeit den Wert-Verlust durch Redundanz zu bestimmen. Dafür kommen neu entwickelte Zähloperationen zum Einsatz, welche diverse Attribute der Elemente gewichten und ins gegenseitige Verhältnis setzen (mehr dazu im Kapitel 5.3 „II: Die Metriken“). Es ist durchaus bekannt, dass viele weitere Faktoren in die exakte Berechnung von Wertverlust durch Redundanz hineinspielen können, jedoch ist dies theoretisch eine geeignete Basis für den weiteren Verlauf des Nachweises. Durch den fehlenden Aspekt der vor dem Nachweis erforderlichen empirischen Ermittlung von Daten ist es schwierig, konkrete Schlussfolgerungen zu ziehen. Wie angesprochen ist es jedoch möglich, diverse Parameter zu nutzen, welche später (bei geeigneten gesammelten empirischen Daten) entsprechend belegt werden können. Ebenso sind weitere Konzepte vorstellbar, welche später im Nachhinein umgesetzt werden könnten, wie die Rückkopplung oder Übertragung von Redundanz über weite Strecken oder das Nutzen von funktionalen Beziehungen zwischen Komponenten. Gerade der letzte Punkt ist dabei etwas hervorzuheben, da dieser eine mehr oder minder wichtige Rolle spielen kann. Ursprünglich wurde angedacht, funktionale

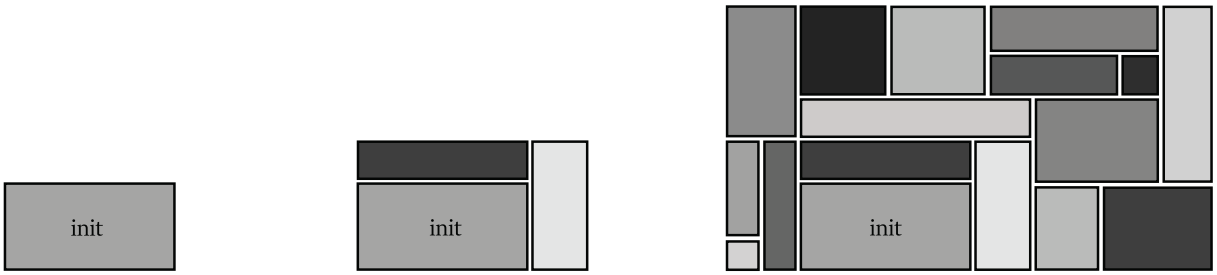


Abbildung 5.8: Die Evolutionstheorie: vom `init`-System zu einem komplexen Softwaresystem

Abhängigkeiten zwischen Komponenten mit zu nutzen, da dies sehr wahrscheinlich einen großen Einfluss auf bestimmte Kostenbereiche hat. Im Graphen wäre dann ein neuer Kantentyp entstanden, welcher am Anfang und Ende gleichermaßen Komponenten besitzt. Allerdings wurde dieser Ansatz verworfen, da dies eine Eigenschaft des Systems darstellt, welche nicht im direkten Bezug zur Datenredundanz steht. Es ist durchaus berechtigt, neue Elemente mit einzubeziehen, welche abseits des eigentlichen Konzepts stehen, jedoch ist eine Abhängigkeit zwischen Komponenten ein Faktor, der zu stark in das Konzept eingreifen würde. Die Wichtigkeit einer Komponente wird bereits durch die Nutzung eines Datenbestandes beschrieben, eine zusätzliche Quelle von Abhängigkeit unter Komponenten würde die Intention des Graphen verfehlen. Sollte sich herausstellen, dass die Komponenten gewichtet werden sollten, ist dies durchaus als neue Annotation möglich, jedoch sollte von einem neuen Kantentyp abgesehen werden, um das Modell auf das zu beschränken, was nachgewiesen werden soll, um am Ende keine Schlüsse zu ziehen, die auf anderen Elementen als Datenredundanz aufgebaut sind. Mit der gleichen Begründung sind viele andere Eigenschaften im Graphen selbst nicht oder nur indirekt vertreten.

5.2.3 Evolution

Die im vorherigen Abschnitt definierten Elemente des Graphen reichen aus, um das Phänomen der *unmanaged Redundancy* zu erklären. Allerdings fehlt bislang die Möglichkeit des Systems, und damit im Graphen, Erweiterungen zu modellieren und den Fortschritt der Evolution im Graphen darzustellen. Wie bereits angesprochen, kann man in Abbildung 5.6 bereits zwei Erweiterungen des Graphen erkennen, *ext1* sowie *ext2*. Dass hierbei die Erweiterungen als einfaches Rechteck hinterlegt werden können, ist der Einfachheit des Graphen geschuldet, sodass bei komplizierten Graphen die Elemente durchaus nicht mehr so einfach als ein Set umrandet werden können. Dabei sind verschiedene Möglichkeiten gegeben, einzelnen Elementen die Zugehörigkeit zu einer bestimmten Erweiterung anzugeben, entweder durch die farbigen Rechtecke, durch die Färbung der Elemente selbst oder durch ein Attribut der Elemente.

Jede Erweiterung, formal als Menge von Elementen mit $\mathbb{E}_i^+$ betitelt, beinhaltet eine Menge an Komponenten $\mathbb{C}_i^+$, eine Menge an Datenbeständen $\mathbb{D}_i^+$ sowie eventuelle Beziehungen $\mathbb{X}_i^+$. Die einzelnen Kanten, welche in der Erweiterung dazu kommen ($\mathbb{X}_i^+$), werden weniger Beachtung finden, da sich die Erweiterungen und die Evolution hauptsächlich auf die Knoten-Elemente im System beziehen. Dabei kann eine Erweiterung entweder bereits vorhandene oder komplett neue Elemente enthalten, welche erweitert (respektive hinzugefügt) werden. Damit kann eine Erweiterung nie „leer“ sein, da eine Erweiterung immer entweder mindestens ein bekanntes Element des Systems erweitert oder ein neues Element dem System hinzufügt. So kommen nach und nach neue Erweiterungen an das System hinzu (siehe Abbildung 5.8). Jedes neu hinzugefügte Element bekommt ein anderes, ungenutztes Label, welches im bisherigen System eindeutig identifizierbar

ist. Neue Kanten können von neuen zu alten Elementen gezogen werden, ebenso umgekehrt. Die sich während einer Erweiterung entsprechenden Werte *Function Points* und *Entities* von älteren Komponenten und Datenbeständen werden einfach neu eingefügt und aktualisiert. Sollte ein (bereits vorhandenes) Element einer erneuten Erweiterung unterzogen worden sein, sollte dies für spätere Metriken notiert werden. Dies ändert aber nichts an der „Ursprungs"-Erweiterung des Elements, welche nie verändert wird im immer die „Entstehungszeit" darstellt.

Die Erweiterung besitzt dabei eigentlich keine Logbuch-Funktion zum Nachvollziehen der Geschehnisse, was wann und wo geändert wurde, sondern dient eher als Zeitachse für die Veränderungen des Systems, da eine normale kontinuierliche Zeitachse für die Veränderungen eines Graphen hierbei nicht praktikabel wäre. Die meisten realen Systeme definieren sich über die Erweiterungen des Systems durch einzelne Projekte, was dafür geeignet ist, dies als Quasi-Zeitachse anzusehen. Die fortwährende Veränderung der einzelnen Attribute der Elemente im Graphen machen es nötig, diese bei jeder hinzugefügten Erweiterung neu zu evaluieren. Die angesprochenen Zähl-Algorithmen rechnen dabei immer über alle bestehenden Elemente und analysieren den Verfall des Wertes durch die sich neu ergebene Situation nach jeder Erweiterung.

Dabei stellt sich am Ende auch die Frage, wie ein geeigneter Graph dieser Form in der Praxis aussieht. Dies ist kein neues Problem und basiert auf dem Forschungsfeld der synthetischen Produktion realitätsgetreuer Netzwerke. Es ist im Rahmen dieser Arbeit nachvollzogen worden, dass diese Art von Graph, wie er hier erstellt wurde, durchaus in der Praxis handhabbar ist und dass die Abstraktionshöhe hoch genug ist, damit die Graph-Semantiken auf Systeme in der Praxis angewendet werden können.

5.3 II: Die Metriken

Nachdem in dem vorherigen Kapitel die grundlegenden Elemente des Graphen, und damit der Darstellung des Nachweises, aufbereitet wurden, folgt nun der Teil der Messbarkeit. Diese ist per Metriken realisierbar. Eine Metrik ist, wie bereits ausführlich im Kapitel 3.4 „Metriken in Softwaresystemen" erläutert, ein direkter oder abgeleiteter (indirekter) Kennwert, der eine bestimmte Eigenschaft in einer Software numerisch darstellt, zum Beispiel die Anzahl aller Codezeilen ($\#LOC$) in einer Komponente. Um die *unmanaged Redundancy* als strukturelle Eigenschaft des Systems in eine Form des Wertes umzuwandeln, benötigt man zuvor die folgenden zwei Schritte:

1. Die *Graph-Metriken* sind dazu da, Eigenschaften aus dem System auf den Graphen zu übertragen. Wie im vorherigen Kapitel beschrieben, kommen bestimmte Elemente im Graphen mit Attributen, welche Werte des Systems widerspiegeln, wie zum Beispiel die Attribute $CRUD$ oder auch $\overline{Rd}$. Diese Werte werden direkt dem System entnommen und auf den Graphen projiziert, sodass eine möglichst praxisnahe Simulation und eine sinnvolle Messbarkeit vorgenommen werden kann. Dabei sollen nun die im vorherigen Kapitel eingeführten Graph-Elemente formal definiert werden.

2. Die *Wert-Metriken* bilden den zweiten Schritt, und extrahieren die Information aus den zuvor gebildeten Graph-Metriken. Dazu werden die Eigenschaften des Graphen sowie dessen Struktur ausgewertet und in neue, abgeleitete Metriken eingebaut. Am Ende steht als Resultat der Wert-Metrik nicht wirklich der Wert des Systems, sondern, spezifisch in dieser Arbeit, der Aufwand oder die Investition in das entsprechende System. Damit der Aufwand anschließend in einen geeigneten Wert des Systems umgewandelt werden kann,

müssen auch die Wert-Metriken ein einzelnes numerisches Ergebnis liefern. Wertet man anschließend die einzelnen Wert-Metriken aus, entsteht das Gesamt-Ergebnis als Wert von Softwarearchitektur durch dessen Struktur, abgebildet durch einen Graphen.

Die Metriken an sich bilden den vergleichsweise neuesten Aspekt in dieser Arbeit. Die *Managed Evolution*, die Redundanz-Arten, der Graph und einiges Anderes sind eher Ableitungen aus bisheriger Forschung und Praxis. Die Metriken, welche im Folgenden vorgestellt werden, basieren auf „Eigenforschung“ und sind, theoretisch gesehen, relevant genug, um eine sinnvolle Aussage über den Wert von Softwarearchitektur zu treffen.

Nach Fenton *et al.* [16] sind Metriken über strukturelle Eigenschaften meistens durch diverse Graphen definiert, einfach weil ein strukturierter Graph sich von Natur aus für die Erfassung strukturierter Eigenschaften eignet. Dafür wurden in dieser Literatur die vier folgenden Eigenschaften als strukturelle Elemente hervorgehoben:

1. *Complexity*, als die Eigenschaft, welche die Komplexität eines Elements angibt. Je komplexer ein System, desto schwieriger und anfälliger wird es.

2. *Length* als die Eigenschaft über die Größe einer Komponente. Das spielt vor allem dann eine Rolle, wenn der Aufwand für eine Entwicklung bestimmt werden soll.

3. *Coupling* ist die Eigenschaft, inwieweit verschiedene Komponenten voneinander abhängen. Das ist besonders dann relevant, wenn *Technical Debt* und *Architecture Erosion* entfernt werden soll. [2] [28]

4. *Cohesion* ist die Eigenschaft, inwieweit verschiedene Elemente im System sinnvoll zusammenarbeiten und zusammengehören. Auch dies ist für strukturelle Analysen wichtig, besonders was das Rearchitecting und Refactoring angeht. Dies kann auch ein einzelnes Element betreffen, inwieweit die Aufgaben, welches es übernimmt, „fokussiert“ sind. [2] [28]

In dieser Arbeit werden diese Eigenschaften öfters vorkommen, wenn auch in anderer Form. Die *Complexity* wird dabei als die Menge an Elementen dargestellt, die *Length* durch die beiden Attribute *Entities* sowie *Function Points*, und *Coupling* durch die Redundanz- und Nutz-Beziehungen. Nur *Cohesion* wird nicht direkt hinzugenommen, da dies bei dieser Thematik der *unmanaged Redundancy* theoretisch keine relevanten Auswirkungen besitzt, denn der Schaden durch *unmanaged Redundancy* ist größtenteils unabhängig davon, inwieweit stark die einzelnen Elemente zusammenhängen. Diese vier Eigenschaften finden sich also auch in dieser Arbeit wieder, jedoch ist es falsch, anzunehmen, dass diese der ausschlaggebende Punkt waren. Vielmehr fanden diese sich im Nachhinein in dieser Arbeit wieder, was am Ende nur die Belastbarkeit der erstellten Metriken und dem zugehörigen Graphen stützt.

Die hier erstellten Graph-Metriken folgen grundsätzlich den Eigenschaften des zu simulierenden Systems. Die Logik hinter den hier erstellten Wert-Metriken folgt dabei ebenfalls einer strengen Linie. Dazu ist in Abbildung 5.9 eine grafische Übersicht gegeben, welche sechs Schritte dabei durchlaufen werden. Zuerst wurde der Graph definiert, wie im vorherigen Kapitel beschrieben. Aufbauend auf diesem Graphen wurden die Eigenschaften der drei Kostenbereiche *Adaptive*, *Corrective* und *Preventive Maintenance* untersucht und entsprechende Einflussfaktoren (Beziehungen und Annotationen) im Graphen gesucht. Diese Faktoren, welche Einfluss auf einen der drei Bereiche haben, wurden dann in entsprechender „Strafmaß“-Manier als Faktor in den Zähl-Algorithmus der Wert-Metrik eingearbeitet. Dabei wurden entsprechende Annahmen getroffen, inwiefern sich diese Einflussfaktoren äußern. Aus der Wert-Metrik entsteht ein numerischer Wert für Reparaturkosten, der als Aufwand für die Behebung der *Technical Debt* zu verstehen ist. Um

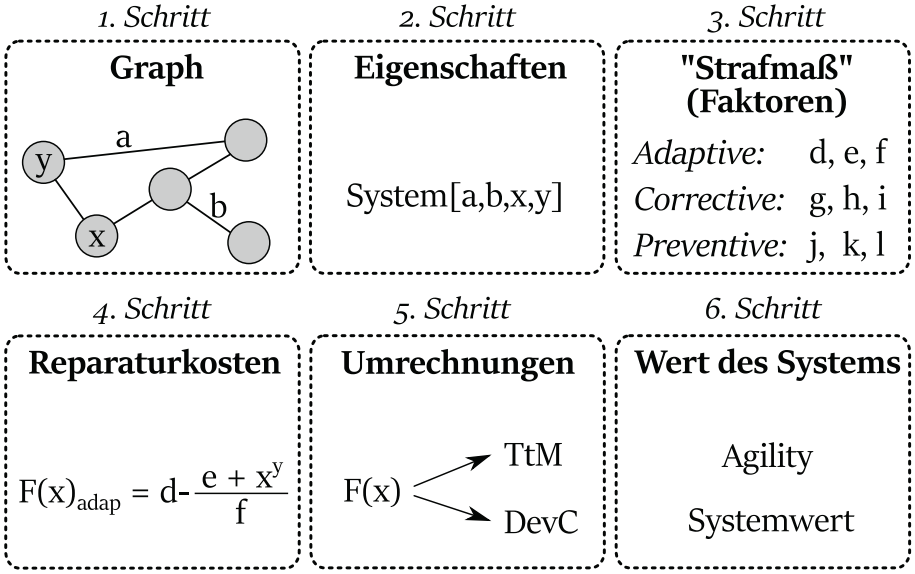


Abbildung 5.9: Die Logik hinter den Wert-Metriken

den echten Wertverlust darzustellen, kann dieser entstehende Reparatur-Wert in die gängigen Werte von *TtM* und *DevC* umgerechnet werden, um damit einerseits die *Agility* des Systems zu bestimmen, und andererseits den Wertverlust in Tagen und Euro zu erhalten.

Die folgenden beiden Abschnitte sollen nun klären, wie die einzelnen Metriken definiert sind, wie man diese anwendet und warum diese entsprechend gewählt wurden.

5.3.1 Graph-Metriken

Die sogenannten Graph-Metriken sind, wie bereits beschrieben, jene Metriken, welche die wesentlichen Eigenschaften des Systems auf den Graphen abbilden. Dabei wurden die Eigenschaften des Systems beachtet, welche auch relevant bezüglich der Hypothese sind. Die entsprechenden Eigenschaften, welche in dieser Arbeit als relevant genug gewertet wurden, sind die Folgenden:

1. die *inhaltliche Größe* der Elemente,
2. der *Umfang der Nutzung* einer Komponente bezüglich einem Datenbestand, und
3. die *Höhe der Redundanz* zwischen zwei Datenbeständen.

Mit diesen drei Eigenschaften ist später der Wertverlust recht gut ermittelbar und auch in der Höhe bestimmbar. Weitere Eigenschaften, welche darauf Einfluss gehabt hätten, wären zum Beispiel die *Relevanz* von Elementen, die *Struktur* von Elementen in deren Inneren, die *Ordnung* oder *Komplexität* der Elemente sowie viele weitere Eigenschaften. Dass diese nicht mit einbezogen wurden, liegt an einem der beiden Punkte: (a) die Eigenschaft kann halbwegs gut durch eine der obigen abgeleitet werden, oder (b) die Eigenschaft ist nicht relevant genug, um den Wert bezüglich der *unmanaged Redundancy* abzuändern.

Die Graph-Metriken sagen, entsprechend ihrem Sinn, zu keinem Zeitpunkt etwas über den Wert des Systems aus. Im Gegensatz zu den Wert-Metriken bilden sie allerdings die Grundlage für die Einordnung der vorliegenden Struktur im System, da letztlich auch der Wert des Systems anhand und durch dessen Struktur ermittelt werden soll. Gemäß den in Kapitel 3.4.1 „Software

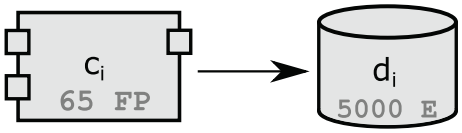


Abbildung 5.10: Die inhaltliche Größe von Komponenten und Datenbeständen

Produkt Attribute“ genannten Kategorien gehören die Graph-Metriken vermutlich am ehesten dem Attribut *Komplexität* aus der Kategorie der *internen* Attribute sowie dem Scope *Structural and complexity metrics* an. *Intern* deswegen, da bei der Erstellung der Graph-Metriken keine aus dem Umfeld entnommenen Messwerte einspielen und nur das System selbst zählt.

Die Inhaltliche Größe Diese Eigenschaft beinhaltet als Einzige mehr als eine Metrik, da sowohl Komponenten als auch Datenbestände eine inhaltliche Größe besitzen. In vorherigen Kapiteln wurde bereits erwähnt, welche das sind: zum einen die *Function Points* (*FP*) für die Komponenten, zum anderen die *Entities* (*E*) der Datenbestände.

Metrik-Idee

Der Sinn hinter dieser Metrik soll sein, einen Bezug zur Größe der Beteiligten herzustellen. Es ist durchaus relevant für den Schaden an einem System, wie groß das eigentliche System ist, da sich durch die Größe auch andere Eigenschaften definieren, wie zum Beispiel Komplexität oder Relevanz. Hierbei spielt die inhaltsbasierte Größe eine besondere Rolle, da damit von dem eigentlichen System abstrahiert werden kann. Eine „echte“ Größe könnte das Problem aufwerfen, dass zu viele verschiedene Möglichkeiten und Programmierstile den wichtigen relativen Bezug zwischen Elementen manipulieren. Der zweite wichtige Punkt ist, dass durch die Definition einer inhaltlichen Größe auch der erste Bezug zu einem Systemwert hergestellt werden kann, indem man ermitteln kann, was ein Unternehmen im Schnitt für ein Element der Größe X ausgibt. Dies ist im Projektmanagement eine durchaus beliebte Methode zur Einschätzung des Aufwandes für das entstehende System.

Es ist das allererste wichtige Merkmal, das hier erwähnt werden sollte: beide inhaltliche Größen sind austauschbar. Die *FP* können mit jeder beliebigen anderen inhaltlichen Größe (zum Beispiel *Object Points* oder *Use Case Points*) getauscht werden, solange dies konsistent im Modell geschieht. Ebenso ist *E* austauschbar gegen andere Größen, die Hauptsache ist, dass der Ersatz auch eine echte Alternative für die inhaltliche Größe darstellt. Das funktioniert bei beiden Eigenschaften deswegen so gut, da in allen Fällen die inhaltliche Größe nur als relativer Wert genutzt wird, und solange dies konsistent im Modell umgesetzt wird, bleibt das Verhältnis immer gleich. Der einzige Zwang, den Wert der inhaltlichen Größe abzuändern, hängt an der Definition einer Komponente. Ist die Komponente im Graphen eine Klasse oder ein Modul, dann sollte es mit *Function Points* keine Probleme geben. Sollte eine Komponente jedoch auf ein Teil des Systems abgebildet werden, der nicht durch eine *FP*-Berechnung gehandhabt werden kann, muss ein Äquivalent eingesetzt werden. Im Gegensatz dazu kann jeglicher Datenbestand immer in *Entities* bemessen werden, da dies eine einfache Metrik für die Anzahl der Informationen darstellt, die jeder Datenbestand per Definition mitbringt. Im Graphen selbst wird die inhaltliche Größe als einfache Annotation innerhalb des entsprechenden Knotens realisiert, wie in Abbildung 5.10 abgebildet.

Die inhaltliche Größe von Elementen spielt in jeder der später folgenden Wert-Metriken eine wichtige Rolle, da nur anhand derer die *TtM* und die *DevC* festgestellt werden kann. Dabei wird sich die strenge Monotonie der inhaltlichen Größe zunutze gemacht, was bedeutet, dass eine Komponente c_a tatsächlich mehr Aufwand als eine andere Komponente c_b benötigt, wenn für die inhaltliche Größe der *Function Points* auch $FP(c_a) > FP(c_b)$ entsprechend gilt. Für Datenbestände d_i gilt dies analog. Dass für einzelne Elemente unterschiedliche inhaltliche Größen benutzt werden, liegt an den grundlegend verschiedenen Arten, den Komponenten und den Datenbeständen. Ein Datenbestand kann nicht in *FP* bemessen werden, umgekehrt wird es schwer, eine Komponente in *E* zu ermessen.

Die inhaltliche Größe *FP* von Komponenten ist hierbei keine eigens erstellte Metrik und ist in der Praxis weit etabliert. Die konkrete Berechnung hängt unter anderem von dem Input, den Berechnungen und den Zuständen der Komponente ab. Eine genaue Beschreibung, wie die *FP* für eine Komponente ermittelt werden, ist in Fenton *et al.* [16] ausführlich beschrieben. Man kann damit additiv auch die *FP* für ein gesamtes Modul berechnen. Da die einzelnen Komponenten im Graphen untereinander weder indirekte noch direkte Beziehungen haben, sind keine weiteren Probleme zu beachten. Somit ist die erste Graph-Metrik geklärt, die der *Funktion Points*, die inhaltliche Größe der Komponenten, bezeichnet mit *FP* und formal in Kurzform dargestellt mit $FP_{c_i} = n$ mit n als die Höhe des spezifischen Wertes für die Komponente c_i.

Für einen Datenbestand berechnet sich die Angabe der *Entities* (die inhaltliche Größe von Datenbeständen) im einfachsten Falle aus der Anzahl an Zeilen in einer Datenbank-Tabelle. Sind Daten direkt in den Quellcode hinein programmiert, dann ist es schwieriger, die Anzahl an Informationen dieses Datenbestands zu ermitteln. Aber dennoch ist es auch da möglich, da eine Information im Quellcode üblicherweise als Attribut oder Variable in `int`, `float`, `double`, `String` und so weiter abgespeichert wird. Das Wichtige bei der inhaltlichen Größe von Datenbeständen ist, wie genau eine *Entity* eines Datenbestands definiert wird. Generell stellen die *Entities* die Anzahl an gespeicherten Informationen dar, sozusagen Informationseinheiten. Schwierig wird es genau dann, wenn eine Informationseinheit (also eine *Entity*) in mehrere Teile aufgesplittet wird, zum Beispiel eine Adresse nicht komplett sondern in Teilen voneinander getrennt gespeichert wird. Ist dann die gesamte Adresse eine *Entity* oder die einzelnen Teile selbst jeweils eine? Im Graphen ist pro Datenbestand nur ein Wert für *Entities* erlaubt, unabhängig der darin befindlichen Struktur, wie zum Beispiel verschiedene Tabellen oder Datenbank-Container. In der Basis dieser Arbeit wurde definiert, dass ein Datenbestand immer dann separat im Graphen abgebildet wird, wenn er unabhängige Informationen besitzt. Das kann bedeuten, dass ein Datenbanksystem im Graphen mehrere Datenbestände ausweist, da er verschiedene, voneinander unabhängige Tabellen besitzt. Dagegen sind zum Beispiel fest einprogrammierte Informationen im Quellcode jeweils für sich selbst eine eigene *Entity*, zusammengefasst in einer Klasse (dem zugehörigen Datenbestand im Graphen), welche dann eine bestimmte Höhe an *Entities* besitzt. Überträgt man dieses „Kapsel-Information-Konzept“ auf die vorher angesprochenen Datenbanken, dann kommt man zu dem Schluss, dass eine Adresse, welche per Fremdschlüssel-Beziehung auf mehrere Tabellen gesplittet ist, als verschiedene Information zählen muss und damit die eine Information, die Adresse, mehrmals zu den *Entities* verschiedener Datenbestände hinzugezählt wird. Das ist insofern nicht falsch, wenn jede Tabelle auch separat zugegriffen, gewartet, und modifiziert werden kann, unabhängig der anderen Tabellen, Tablespaces oder Datenbank-Container. Im Endeffekt bilden solche Fälle neue synchronisierte Redundanzen im Graphen zwischen den Datenbeständen bei den Teilen der Information, welche den Fremdschlüssel beinhalten um aufeinander zu referenzieren. Bei der Frage nach den *Entities* ist es daher relevant, welche Informationen im Verbund betrachtet werden und welche eher einzeln benutzt werden.

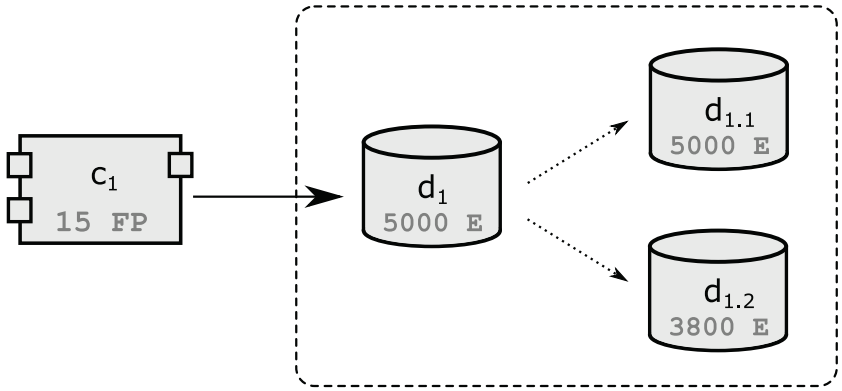


Abbildung 5.11: Ein Beispiel, in dem gesplittete Informationen als eine *Entity* zählen

Dies ist insofern nötig, um den echten Informationsgehalt eines Datenbestandes feststellen zu können, ohne zu sehr von dessen Implementierung abhängig zu sein.

Es gibt keine feste Regel, nach denen eine *Entity* festgelegt werden kann, da dies ein eher abstraktes Gebilde ist und dazu dient, die Anzahl an Informationseinheiten als inhaltliche Größe zu erfassen. Es existieren nach dieser Auslegung der Definition von Datenbeständen und deren *Entities* nur noch sehr wenige Beispiele, in denen verschiedene und voneinander getrennte Teile einer Information als immer noch eine *Entity* zählen. Ein Beispiel hierfür ist die Aufsplittung einer Adressen-Tabelle in mehrere Tabellen, welche jedoch immer nur von der Haupt-Tabelle referenziert werden (siehe hierzu Abbildung 5.11). Damit würde das restliche System nur mit der Haupt-Tabelle interagieren und obwohl die Informationen auf viele Tabellen gesplittet sind, ist die Anzahl der *Entities* nur die Anzahl aller Zeilen in der Haupt-Tabelle. Das funktioniert natürlich nur, wenn keine der Unter-Tabellen von woanders noch benutzt wird und lediglich der Haupt-Tabelle dienen, da sonst auch aus den Unter-Tabellen Redundanzen entstehen können, was dazu führt, dass diese einen eigenen Datenbestand begründen. Die Annahme, dass die Höhe an *Entities* einer Datenbank-Tabelle (und damit einem Datenbestand im Graphen) immer genau die Anzahl an Zeilen darstellt, ist in den meisten Fällen dadurch richtig. Ein Fall, in dem dies jedoch nicht so ist, ist zum Beispiel der, wenn aus irgendeinem (implementationstechnischen) Grund diverse Zeilen immer als Paare auftreten, dann halbiert sich die Anzahl der *Entities* und damit die der „wahren“ Informationseinheiten. Damit ist die zweite Graph-Metrik umrandet: die inhaltliche Größe von Datenbeständen, betitelt mit *Entities*, bezeichnet mit E, formal in Kurzform beschrieben durch $E_{d_i} = n$ mit n als spezifischer Wert für die *Entities* des Datenbestands d_i im Graphen.

Umfang der Nutzung Der Umfang der Nutzung ist der zweite Faktor neben der inhaltlichen Größe, der eine gewisse Relevanz ausmacht. Im Gegensatz zu der inhaltlichen Größe, was eine Annotation der Knoten ist, ist der Umfang der Nutzung eine Annotation einer Kante, im Speziellen ein Attribut einer „Nutz-Kante“ von einer Komponente hin zu einem Datenbestand.

Metrik-Idee

Der Umfang der Nutzung gibt wieder, welche Auswirkungen Komponenten auf einen Datenbestand haben. Eine große Komponente besitzt nicht automatisch viel Relevanz in Bezug auf Datenredundanz, wenn diese fast nicht in das Geschehen eingreift, und umgekehrt ana-

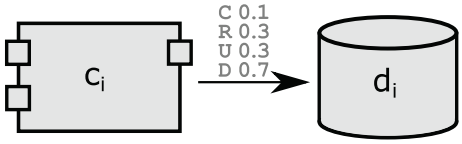


Abbildung 5.12: Die Nutzung einer Komponente auf einem Datenbestand

log. Zudem war die Intention hinter dieser Metrik, nicht nur eine Höhe der Verwendung anzugeben, sondern direkt eine Fallunterscheidung einzubauen. Der Sinn dahinter ist, dass nicht nur die Höhe der Verwendung eine Rolle spielt, sondern auch die Verwendungsart verschiedene Auswirkungen auf die negativen Folgen der Datenredundanz aufweisen kann. Dabei erscheint es logisch, dass ein Schreib-Zugriff auf Daten eher einen Schaden anrichten kann, als ein reiner Lese-Zugriff. Deswegen wird diese Unterscheidung ebenfalls in die Metrik einfließen.

Die Nutzung stellt die Relation dar, inwieweit eine Komponente am Geschehen im System bezüglich Redundanz beteiligt ist. Eine riesige Komponente, welche nur marginal auf einen von Redundanz betroffenen Datenbestand zugreift, hat unter Umständen weniger Einfluss auf den Wert der Struktur, als eine andere 10-Zeilen-Komponente, welche jedoch komplett mit dem System interagiert. Um dieses Verhalten zu modellieren, wurde die Nutz-Kante eingeführt, welche immer ihren Anfang bei einer Komponente besitzt und in einem Datenbestand endet. Dies gibt die erste Information, inwieweit eine Komponente mit dem System verbunden ist. Dabei kann eine Komponente mehrere ausgehende Nutz-Kanten haben und ein Datenbestand kann mehrere eingehende Kanten beinhalten. Der einzige Sonderfall, welcher nicht erlaubt ist, sind parallele Nutz-Kanten, welche jeweils die gleiche Komponente mit dem gleichen Datenbestand verbinden, da jede Komponente entweder insgesamt oder gar nicht auf einen Datenbestand zugreift.

Wie in dem vorherigen Kapitel zu dem Graphen bereits erläutert, ist der Kernpunkt der Nutz-Kante die sogenannte *CRUD*-Metrik, was für die vier verschiedenen Fälle der Nutzung von Daten stammt: *Create*, *Read*, *Update* und *Delete*. Damit ist der Umfang der Nutzung von Daten recht genau umrandet, weil man mit diesen vier Operationen fast alle Datenmanipulationen durchführen kann. Diese vier Werte werden zusammen mit ihren eigenen Werten an die Nutz-Kante als Vektor hinzugefügt, wie in Abbildung 5.12 aufgezeigt.

Bleibt natürlich noch die Frage, woraus sich die einzelnen Werte für die vier Möglichkeiten zusammensetzen. Wenn der Umfang der Nutzung festgestellt werden soll, muss darauf geachtet werden, inwieweit der Umfang der Komponente eingeschätzt werden kann. Die erste Möglichkeit ist eine Basisberechnung über die Anzahl von Methoden. Man könnte dabei bestimmen, wie viele Methoden eine Mechanik implementiert haben, welche den Datenbestand nutzen. Während das auf den ersten Blick als valider Ansatz erscheint, erkennt man bei genauerer Betrachtung jedoch, dass dies etwas ungünstig gewählt ist. Das erste Problem wäre, dass man nicht davon ausgehen kann, dass alle Komponenten, universell wie sie sein sollen, nicht zwangsweise Methoden im klassischen Sinn aufweisen, je nachdem, was von dem System auf eine Komponente im Graphen abgebildet wird. Das zweite Problem wäre, dass die Anzahl der Methoden eine unzuverlässige Größe ist, was den Programmierstil betrifft. Besonders bei plattformübergreifender Programmierung kann die Anzahl der Methoden pro Funktionalität extrem variieren. Ein Beispiel dafür ist folgende Situation: eine Java-Klasse A benutzt das Hibernate-Framework zusammen mit einigen

generischen Klassen, um variabel auf einzelne Werte einer Datenbank zuzugreifen. Das macht in der Summe für die Klasse *A* vielleicht etwa fünf Methoden-Aufrufe für das Erlangen eines Wertes, ohne interne Methoden vom Hibernate-Framework. Die Klasse *B* benötigt ebenfalls nur einzelne Werte, nutzt aber nur nativ die JDBC-Treiber. Das macht in der Summe weniger Methoden für die gleichen Werte. Übrig bleibt der Eindruck, dass *A* weniger Nutzung an der Datenbank besitzt als *B*, was jedoch falsch ist, da beide im Endeffekt die gleichen Werte haben möchten. Aber auch die einfachste Messung über die Anzahl der Codezeilen, welche auf eine Datenbank zugreifen, ist mit dem Szenario gleichzusetzen und ebenfalls ungeeignet. Damit fällt die Messung über die benutzten Methoden und $\#LOC$ weg und es verstärkt sich der Ansatz, etwas abstrakteres zu nutzen, im besten Falle etwas Unabhängiges von der benutzten Implementierung und Plattform. Es wurde sich entschieden, die *Use Cases* als Wert für die Nutzung zu nehmen, denn ein *Use Case* ist universell im Software-Design verankert und kann jederzeit hinzugezogen werden. Wenn man nun den Zugriff auf einen Datenbestand mit den *Use Cases* kombiniert, kommt man auf ein Verhältnis, welches unabhängig der Implementierung und Plattform ist. Die Anzahl der Zugriffe aller *Use Cases* wurde somit als Grundlage des Umfangs der Nutzung gewählt. Eine Alternative wäre die Nutzung von *Workflows* innerhalb der Komponente, jedoch erwiesen sich die *Use Cases* als geeigneter, da diese einerseits bereits aus der Analyse- oder Design-Phase des Systems vorhanden sein sollten und zudem mehr mit der inhaltlichen Größe zusammen passen, da *Workflows* durchaus sehr unterschiedlich ausfallen können und in sich mehrere verschiedene Aufgaben (beziehungsweise Datenbankzugriffe für verschiedene Zwecke) gebündelt haben könnten, was bei den *Use Cases* genauer erfasst wird.

Bleibt noch die zweite Frage offen, inwieweit man das erstellte Verhältnis durch die *Use-Cases* als Graph-Metrik einbauen möchte. Diese Frage beinhaltet besonders die folgenden Punkte:

- Spielt die Zeit eine Rolle? (Zugriffszeiten, Rechenzeit pro Use Case, ...)
- Spielen Mechaniken eine Rolle? (Caching, Synchronisation, Lazy/Eager Loading, ...)
- Inwieweit ist es relevant, was genau von der Datenbank benutzt wird?

Nach einiger theoretischer Überlegung können die drei Punkte wie folgt beantwortet werden. Die Zeit spielt keine Rolle, da die *unmanaged Redundancy* nicht wirklich Zeit-abhängig ist, was den Zugriff auf Daten anbelangt. Entweder es liegt eine hinreichende Synchronisierung der Redundanz vor oder nicht. Und eine nicht hinreichend synchronisierte Redundanz ist laut Annahme in diesem Nachweis bereits schädlich. Aus dem gleichen Grund spielen die benutzen Zugriffs-Mechaniken keine große Rolle.Die Frage, ob beides auf die Höhe des Wertverlustes wirkt, konnte nicht hinreichend theoretisch belegt werden, sodass dieser Fakt nicht beachtet wurde. Dadurch definiert sich der Wert eher durch die inhaltliche Größe als durch die genaue Art und Weise der Nutzung. Bleibt noch der letzte Punkt, inwieweit sich eine partielle Nutzung des Datenbestandes von einer kompletten Nutzung unterscheidet. Dies ist der erste wirkliche Punkt, an dem das Modell verfeinert werden kann, da es durchaus einen Unterschied machen könnte, ob eine Komponente auch den redundanten Teil des Datenbestandes tatsächlich anfasst oder nicht. In dem Graphen dieses Nachweises wurde dieser Fakt vernachlässigt, da dies theoretisch nicht die Haupt-Quelle des entstehenden Wertverlustes darstellt. Jedoch kann, und das ist der Variabilität des Graphen geschuldet, der entstandene normale *CRUD*-Vektor nochmals mit einem Faktor λ gewichtet werden, um die Relevanz der benutzten Datenteile zu simulieren. Da λ aber auf empirischen Messungen beruht, wird dieser Faktor in dieser Arbeit nicht weiter betrachtet.

Annahme 8
Die Zeit und Art des Zugriffs spielen für die unmanaged Redundancy als Wertverfall keine Rolle. Ebenso ist der konkrete Teil eines Datenbestandes, welcher Redundant ist, nicht relevant.

Der letzte Teil, welcher hinsichtlich der Nutzung noch geklärt werden muss, ist die Entstehung der einzelnen Werte im *CRUD*-Vektor. Dafür dient die einfache Metrik, wie viele *Use Cases* bei jeder der vier *CRUD*-Kategorien tatsächlich den Datenbestand anfragen. Wenn man die Notation $\overrightarrow{\varphi} = \overrightarrow{CRUD}$ mit $\varphi_C, \ldots, \varphi_D$ als einzelne Vektor-Werte nutzt und |Use Case| als Menge aller *Use Cases* einer Komponente betrachtet, ergibt sich insgesamt folgender formaler Ansatz für die Höhe der einzelnen Werte im Vektor:

$$\text{Für } k = C, R, U, D : \varphi_k = \frac{|\text{Use Case}|_k}{|\text{Use Case}|}$$

Hier kann der vorher angesprochene λ-Wert hinzu kommen, welcher zur Relativierung der einzelnen Werte genutzt werden kann, wenn dies später von Nutzen sein sollte. Sind diese Werte einmal definiert und festgelegt, sind die *Use Cases* nicht mehr notwendig. Das bedeutet insbesondere, dass es hinterher nicht mehr nachvollziehbar ist, warum bestimmte Werte im Vektor eine gewisse Höhe haben, selbst wenn diese Frage nicht relevant ist. Das macht die *CRUD*-Werte, also den Vektor an sich, zu einem Wert, welcher, einmal abstrahiert, nicht so einfach zurückzuführen ist, vergleichbar mit einem Hash-Wert. Im Grunde dient das der Einfachheit des Nachweises, speziell der Darstellung und der darauffolgenden Messbarkeit, da es sich mit einzelnen zusammengefassten Werten (das heißt, einem einzigen klaren Metrik-Wert für eine Eigenschaft) besser handhaben lässt.

Wie man bereits in der Abbildung 5.12 erkennen konnte, kann die Summe aller Einzelwerte größer als eins werden, was daran liegt, dass jeder der Werte φ_k einen Wert zwischen 0 und 1 zugewiesen bekommt, je nachdem, wie viele *Use Cases* die entsprechende Kategorie nutzen. Dies ist jedoch kein Problem und gibt sogar einen Vorteil bei der späteren Auswertung des Wertes, wie im Kapitel 5.3 „II: Die Metriken“ dargestellt wird, da später der *CRUD*-Vektor φ noch dahingehend normalisiert wird, dass auch die Summe seiner Werte wieder zwischen 0 und 1 liegt. Damit ist die Nutz-Kante mit *CRUD* an sich definiert, ab hier kürzer für weitere Formeln mit $\varphi_{c_i d_j}$ als Nutz-Kante zwischen der Komponente c_i und dem Datenbestand d_j notiert.

Höhe der Redundanz Es bleibt noch eine Graph-Metrik übrig, die der Redundanz-Pfeile, welche in Abbildung 5.13 dargestellt sind. Die Redundanz-Beziehung ist die Kern-These der Darstellung und beinhaltet die Metrik, welche Aussagen darüber liefert, inwieweit ein Datenbestand von anderem kopiert hat. Durch die drei benannten Typen von Redundanz, Standard, vererbt und synchronisiert, kann man sehr viele Fälle der *managed* und *unmanaged Redundancy* abdecken.

Metrik-Idee

Die Redundanz wird benötigt, um den Nachweis über eben jene Auswirkungen der Redundanz führen zu können. Ohne einen Wert für Redundanz fehlt sozusagen die Basis dafür. Die Überlegung zur Erstellung dieser Metrik war, eine Quantifizierung der Redundanz auf einen einzelnen numerischen Wert vorzunehmen, und gleichzeitig die verschiedenen Arten

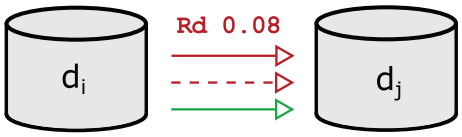


Abbildung 5.13: Die Beziehung der Redundanz: Standard, vererbt und synchronisiert

von Redundanz beizubehalten. Dafür bietet der erstellte Graph mit seiner Kantentypisierung eine geeignete Grundlage. Der nächste Punkt, der in der Intention dieser Metrik liegt, ist die Verwirklichung der Vernetzung von Datenbeständen durch die gerichteten Kanten des Graphen. Das dadurch entstehende „Daten-Netz“ spiegelt dann ein Verhalten im System wider, welches den Schaden im System darlegen soll. Da die komplette spätere Wert-Einschätzung auf dem Wert der Redundanz liegt, muss diese Metrik besondere Details der Redundanz erfassen können. Die Annahme, welche dabei natürlich getroffen werden muss, ist, dass alle Redundanzen bekannt sind, was besonders bei der *unmanaged Redundancy* so gut wie niemals der Fall sein wird. Jedoch kann dieser Fakt bei der Theorie dieser Arbeit außer Acht gelassen werden, da es um den entstehenden Schaden geht, nicht etwa um das Auffinden.

Anfangs ist es unerheblich, welcher Typ von Redundanz zur Anwendung kommt, da der Wert dieser in der Darstellung eine wesentlich höhere Rolle spielt. Dieser Wert, die Höhe der Redundanz, ist dabei allen Typen gleich. Nun stellt sich die Frage, wie man das Kopieren von Werten auf einen einzelnen, möglichst sinnvollen Wert herunter bricht, dass sowohl das Backup-System mit seiner kompletten Spiegelung aller anderen Daten, als auch die hart einprogrammierte ID im Quellcode kausal als Redundanz-Wert erfassbar ist. Anders als bei den Metriken für die inhaltliche Größe (FP und E) und den Umfang der Nutzung (φ) ist die Metrik für die Redundanz eine nach oben und unten limitierte Größe. Das erwähnte Backup-System sollte möglichst den höchsten Wert an Redundanz besitzen, was nur dann möglich ist, wenn es ein oberes Limit für die Redundanz gibt. Es bietet sich also an, eine prozentuale Skala zu nutzen, bei der 100% das obere Limit, und damit die volle Redundanz, darstellt. Abgesehen davon ist eine Mechanik erforderlich, welche auf sinnvollem Weg sicherstellt, dass 8%-ige Redundanz nicht nur numerisch sondern auch tatsächlich weniger Redundanzen besitzt als eine 15%-ige Redundanz.

Die in dieser Arbeit vorgestellte Lösung für das Problem basiert auf der Theorie, dass eine Informationseinheit, eine *Entity*, solange teilbar ist, bis nur noch einzelne *Frames* (Felder) der Information übrig bleiben. Diese Felder sind in einem Datenbanksystem meist tatsächlich einzelne Felder, das heißt, jede einzelne Column-Zelle in einer Zeile ist ein nicht weiter teilbarer Abschnitt der Information. Diese Art der Teilung von Informationen ist aber auch auf andere Systeme anwendbar, denn auch wenn viele nicht wie ein Datenbanksystem organisiert sind, wäre es theoretisch möglich, alle in einem Textdokument enthalten Informationen (zum Beispiel CSV-Codiert) ganz aufzusplitten. Bei hart einprogrammierten Informationen ist es ebenfalls einfach, diese zu unterteilen, da die meisten davon bereits aufgeteilt sind (zum Beispiel Einzelwerte wie eine ID in einer Konstanten).

Beispiele dafür finden sich in Tabelle 5.1 und Quellcode [5.1]. Man kann bei beiden Beispielen erkennen, dass jeweils ein Kunde abgespeichert wurde. In der Datenbank, welche die Tabelle repräsentieren soll, ist eine *Entity* als Kunde abgespeichert, die acht kleinere Informationen der verschiedenen Art in den einzelnen Spalten enthält. In der Java-Klasse sind dagegen diverse Variablen einprogrammiert, welche ebenfalls die Attribute eines Kunden erfassen. Der erste Unterschied liegt dabei auf der *Entity* selbst. Während eine ganze Informationseinheit in

id	f_name	l_name	tel	adr	reg	hungry	notes
42	Cave	Johnson	1234 5678	ABCD	19.04.2011	×	todo

Tabelle 5.1: Das Aufsplitten einer *Entity* in einem Datenbanksystem

der Datenbank als Zeile angesehen werden kann, sind die einzelnen Attribute jeweils selbst eine *Entity*. Dass die Definition einer *Entity* etwas ungenau ist, wurde bereits weiter oben bei der inhaltlichen Größe beschrieben. In der Java-Klasse wäre es daher falsch, anzunehmen, dass die ersten acht Attribute die Information wären, was dahingehend nicht stimmt, dass sie keine feste Information speichern und maximal etwas zugewiesen bekommen. Die wirklichen Informationen in der Klasse stecken in den Statements. Das erste davon ist jenes, welches der `id_cave` den Wert 42 zuweist. Dies ist eine Informationseinheit, also eine *Entity*. Wie man anhand des ersten `if`-Statements sehen kann, werden andere Werte (hier die normale `id` mit dem Wert `id_cave` der *Entity*) verglichen. Die zweite und dritte *Entity* dieser Klasse entstehen durch das zweite `if`-Statement, in dem spezielle Werte für `f_name` und `l_name` zugewiesen und überprüft werden. Es ist eine Ansichtssache, ob beide als *Entities* zählen, da hier ja der Wert der *Entity* zu Vergleichszwecken gesetzt wird. Jedoch wird hierbei Information in Codeform gespeichert, welche durchaus der *Technical Debt*, beziehungsweise den Folgen der *Unmanaged Redundancy*, ausgesetzt ist. Denn sollte sich der Wert `f_name` irgendwann einmal von „Maxine“ auf etwas Anderes ändern, ist die Information hier veraltet und damit ein weiterer Faktor für die *Corrective Maintenance* entstanden. Bleibt die letzte Frage, warum das dritte `if`-Statement nicht als eine *Entity* zählt. Das liegt daran, dass hierbei ein funktionaler Aspekt angewendet wird, nämlich das Aufrufen der Methode `eat()`, wenn der Kunde hungrig ist (`hungry == true`). Soweit unterscheidet es sich zwar nicht von dem zweiten Statement, in dem auch eine Methode aufgerufen wird, wenn `f_name` und `l_name` einen bestimmten Wert haben, jedoch ist das `if(hungry)` unabhängig der speziellen Information und trifft auch weiterhin zu, weil dies eine Mechanik der Klasse ist, und keine direkte Abhängigkeit von einem spezifischen Objekt ist. Das ist natürlich alles Auslegungssache, jedoch ist es zumindest für diesen Nachweis unerheblich, was genau alles als *Entity* definiert wird, weil am Ende der Gesamtwert entscheidend ist und damit die Abstraktionshöhe viel höher liegt als auf der Ebene des Quellcodes.

Sind einmal die *Entities* definiert, dann lassen diese sich wie beschrieben in die *Frames* unterteilen. Das Beispiel aus der Datenbank besitzt damit eine *Entity* mit acht *Frames* (also insgesamt $1 \times 8 = 8$ Frames), die Java-Klasse besitzt dagegen drei *Entities* mit jeweils einen *Frame* (also $3 \times 1 = 3$ Frames). Natürlich kann auch eine Java-Klasse diverse Mechanismen aufweisen, welche sich darin äußern, dass Attribute mit weniger *Entities* und dafür mehr Frames aufweisen, zum Beispiel das Benutzen einer `HashMap`, in der Datenbank-ähnlich Wertepaare gespeichert sind. Diese speichert dann viele *Entities*, mit jeweils mindestens zwei *Frames* pro *Entity*. Die Anzahl der bereitgestellten *Frames* ist stark abhängig der Speicherform, was durchaus nachvollziehbar ist. Aber das Zerteilen der *Entity* auf die einzelnen *Frames* erlaubt eine klarere Definition von Redundanzen im System, wohingegen das Beibehalten von der Anzahl der *Entities* immer noch der kritische Punkt bei der Einschätzung der inhaltlichen Größe von Datenbeständen ist. Da das Ganze natürlich auch ein eher theoretisches Konstrukt ist, wird keiner dazu angehalten, in seinem System die einzelnen *Entities* und *Frames* zu zählen, was sehr schnell unmöglich werden kann.

Diese vollkommene Aufteilung in *Entities* und *Frames* macht es möglich, abstrakt genug über den Inhalt von Informationen Aussagen zu geben sowie eine geeignete Metrik für die Redundanz

```
1  public class MyClient(){
2    private int id;
3    private String f_name;
4    private String l_name;
5    private double tel;
6    private String adr;
7    private Date reg;
8    private boolean hungry;
9    private String notes;
10
11   public void doSomething(...){
12     private int id_cave = 42;
13     ...
14     if(id == id_cave){ lemon(); }
15     if(f_name == "Maxine" && l_name == "Caulfield"){ selfie(); }
16     if(hungry){ eat(); }
17     ...
18   }
19 }
```

Quellcode 5.1: Das Aufsplitten einer *Entity* in einer Java-Klasse

zu erstellen. Die Höhe der Redundanz wird dabei für alle drei Typen der Redundanz gleich berechnet. Dabei liegt das Hauptaugenmerk auf den kopierten Daten in den einzelnen *Frames*. Das bedeutet, sobald ein Datenbestand auch nur einen *Frame* inhaltlich übernimmt, dann ist bereits eine gewisse Redundanz vorhanden. Dass diese durchaus notwendig sein kann, steht außer Frage, jedoch muss diese Redundanz dann hinreichend synchronisiert werden. Die Fälle aus dem Beispiel zeigen auf, wie man bestimmte Werte einfach kopieren kann. In einer Datenbank ist es das Einfachste, den Inhalt bestimmter *Frames* auszulesen und woanders abzuspeichern, durchaus auch mit anderen Namen. Im Quellcode kann man Redundanzen einbauen, indem man die *Entities* im Quellcode mit Werten belegt, zum Beispiel für die Überprüfung nach den speziellen Werten wie `if(f_name == "Maxine")`, welche eigentlich dynamisch referenziert werden können und sollten. Dies ist ein Fall der sogenannten Shortcuts beim Entwickeln der Software, weil es immens an Zeit (und damit Geld) einspart, wenn man den Wert direkt abprüft, anstatt diesen sauber per Interface zu referenzieren. Natürlich kann es notwendig sein, diesen Shortcut zu nutzen, zum Beispiel aus Performance-Gründen, jedoch muss dabei für eine hinreichend gute Synchronisierung gesorgt werden, damit dies nicht in „Vergessenheit gerät“. Das Schwierige an der Duplizierung der Daten in den *Frames* ist, dass diese unter Umständen nicht mehr auffindbar sind, zum Beispiel wenn Spalten in einer Datenbank anders benannt sind, diese jedoch kopierte *Frames* aus anderen Quellen beinhalten. Diese unbekannte *unmanaged Redundancy* ist dann besonders schädlich, da ein großer Mehraufwand erforderlich ist, sie zu finden, sofern sie überhaupt auffindbar ist.

Formalisiert man nun den Redundanz-Pfeil im Graphen, dann muss dabei auf auf die Quelle und auf das Ziel gleichermaßen eingegangen werden. Es erscheint eine einfache aber effiziente Methode am geeignetsten: das Verhältnis der kopierten *Frames* aus der Quelle hin zu dem Ziel. Das Verhältnis wird bestimmt durch die jeweils maximalen verfügbaren *Frames* beider Seiten

und den tatsächlich kopierten *Frames*. Ein weiteres Beispiel soll dies verdeutlichen. Wenn eine neue Datenbank mit insgesamt 100 *Frames* ganze 20 *Frames* aus einer anderen Datenbank mit insgesamt 1900 *Frames* kopiert, dann ist das Redundanz-Verhältnis der kopierten Werte bei

$$Rd = \frac{20}{100} \cdot \frac{20}{1900} = \frac{1}{5} \cdot \frac{1}{95} \approx 0.021$$

Dies scheint auf den ersten Blick wenig, immerhin wird ein Fünftel der neuen Datenbank aus kopierten Werten bestehen. Das Gegenargument ist, dass ein Backup-System, welches nur aus kopierten Werten besteht, keine 100% Redundanz aufweisen kann, wenn die Quelle noch weitere Daten enthält, welche nicht kopiert wurden. Andersherum, besteht ein Ziel aus mehr als den kopierten Daten und wurden jedoch alle Daten aus der Quelle kopiert, dann liegt die Redundanz bei ebenfalls <100%, da nicht alle Werte im Ziel nur kopiert sind sondern auch teilweise echte Neuinformationen darstellen. Damit ergibt sich, dass eine *unmanaged Redundancy* mit nur $Rd = 0.021$ durchaus im Bereich des Sichtbaren ist. Mit dieser Graph-Metrik ist ein Redundanz-Wert von 0 bis 1 möglich, wobei 0 keine Redundanz und 1 eine exakte Kopie der Quelle darstellt. Um eventuellen Verwirrungen entgegenzuwirken, kann man dieses Redundanz-Verhältnis auch eher als die „Gleichheit“ zweier Datenbestände ansehen, in der neben den kopierten Daten (dem Inhalt) auch die Struktur der Datenbestände (die Anzahl der *Frames*) eine Rolle spielt. Nur wenn zwei Datenbestände exakt gleich sind, dann haben diese auch ein *Rd*-Verhältnis von 1.

Um die Berechnungsgrundlage der Höhe der Redundanz formal zu definieren, sei hier folgende Notation genannt, welche die „Gleichheit“ von Inhalt und Struktur vereint

$$Rd_{d_q d_z} = \frac{cF_{d_q}}{F_{d_q}} \cdot \frac{cF_{d_z}}{F_{d_z}}$$

mit d_z als Ziel-Datenbestand, d_q als Quell-Datenbestand, cF als Anzahl kopierter *Frames* und F als gesamte Anzahl der vorhandenen *Frames*. Das *Rd* wird zu einem $\overrightarrow{Rd}$ mit einem roten Pfeil, das heißt einer Standard-Redundanz, wenn die Redundanz nicht synchronisiert ist. Dabei gilt jedoch noch der Hinweis, dass eine vererbte Redundanz $\widehat{Rd}$ keine eigene *Rd*-Metrik besitzt sondern nur in Verbindung mit einer synchronisierten Redundanz einhergeht. Eine weitere Besonderheit ergibt sich, wenn ein *Frame* $f \in cF_{d_z}$ mehr Informationen beinhaltet, als von d_q kopiert werden. Dies führt dazu, dass f weiter in $f_1, f_2 \in cF_{d_z}$ zerlegt werden muss. Ein Beispiel wäre, wenn $f = Name$ mit $f \in cF_{d_z}$ den ganzen Namen darstellt, aber von d_q nur der Vorname redundant kopiert wäre. Dann muss f in $f_1 = Vorname$ und $f_2 = Nachname$ geteilt werden, da anschließend f_2 (im Gegensatz zu f_1) nicht redundant wäre. Dies ist für nachfolgende Berechnungen wichtig.

Um den Wert einer vererbten Redundanz zu bestimmen, wird der Wert der damit verbundenen synchronisierten Redundanz genutzt. Dazu wird eine neue berechenbare Eigenschaft im Graphen benutzt, welche jeder Datenbestand besitzt: die *Fremdheit* σ. Dass ein Datenbestand als Summe all seiner „echten“ Redundanzen (also alles außer vererbte Redundanzen) nicht höher als 1 besitzen kann, ist demnach durch die oben genannte formale Umsetzung erreicht: es gibt nur endlich viele F_{d_z}, es gilt immer $cF_{d_z} \leq F_{d_z}$, und jedes $f \in cF_{d_z}$ kann nur einmal redundant "belegt" werden. Somit ergibt sich immer ein Wert zwischen 0 und 1 als Fremdheit σ, welcher als Summe aller „echten“ *Rd*-Redundanzen eines Datenbestandes angibt, ob dieser entweder komplett eigene Inhalte besitzt ($\sigma = 0$), zu Teilen mehreren anderen Datenbanken gleicht ($0 < \sigma < 1$) oder komplett aus fremden Inhalten besteht ($\sigma = 1$) und damit keinerlei Neuinformation bereitstellt. Um nun den schadhaften Anteil der Fremdheit zu erhalten, wird die Summe aller Standard-Redundanzen gebildet (also σ ohne synchronisierte Redundanzen) und als σ' notiert. Dieses σ' wird dann dazu genutzt, die vererbte Redundanz wie folgt zu berechnen:

$$\widehat{Rd}_{d_i d_j} = Rd_{d_i d_j} \cdot \sigma'(d_j)$$

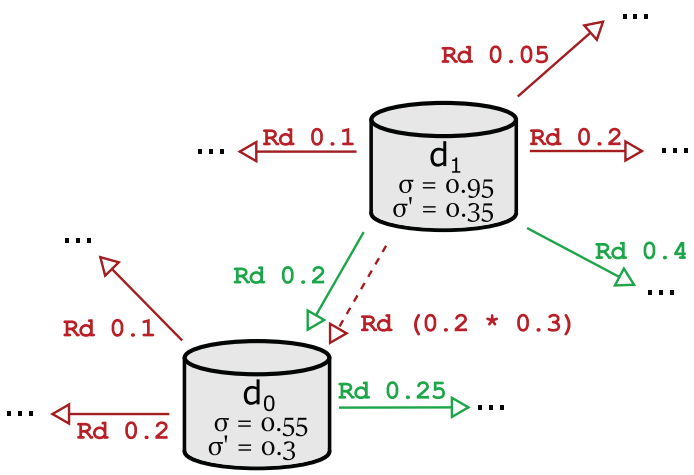


Abbildung 5.14: Die Fremdheit eines Datenbestandes σ und σ'

Ein Beispiel hierfür findet sich in Abbildung 5.14, wo einmal eine einfache Berechnung der Fremdheit aufgeführt wird:

$$\sigma(d_1) = 0.1 + 0.05 + 0.2 + 0.4 + 0.2 \ = 0.95$$
$$\sigma'(d_1) = 0.1 + 0.05 + 0.2 = 0.35$$

Zu beachten ist, dass die „neu entstehende“ vererbte Redundanz (wie auch alle anderen vererbten Redundanzen) nicht mit in σ' hinein zählt, sondern weiterhin ausschließlich Standard-Redundanzen für σ' benutzt werden. Durch die Berechnung von σ' pro Datenbestand ist es möglich, den oben genannten Wert für eine vererbte Redundanz herzustellen, indem der Wert der synchronisierten Redundanz mit σ' des entsprechenden Ziel-Datenbestandes multipliziert wird. Dabei gibt das σ' an, wie viel wirklich schadhafte Redundanz durch die Synchronisation tatsächlich mitgenommen werden kann, mitunter ohne dass der Entwickler sich dessen bewusst ist. Im Beispiel der Abbildung 5.14 wird damit die vererbte Redundanz berechnet durch

$$\widehat{Rd}_{d_1 d_0} = Rd_{d_1 d_0} \cdot \sigma'(d_0) = 0.2 \cdot 0.3 = 0.06$$

Dies garantiert wieder ebenfalls ein $0 < \sigma < 1$ in dem Ziel-Datenbestand, da der neue Rd-Wert der vererbten Redundanz $\widehat{Rd}_{d_1 d_0}$ nicht in $\sigma(d_1)$ eingeht, sowie $\sigma'(d_1)$ ebenfalls nur kleiner werden kann, da

$$\widehat{Rd}_{e'_i} \leq Rd_{e'_i}$$

durch die Multiplikation bei Zahlen kleiner 1 immer gilt. Es ist möglich, die Fremdheit σ und die schadhafte Fremdheit σ' im Voraus der Metriken zu berechnen und als Annotation an die Knoten der Datenbestände im Graphen anzuhängen, was jedoch nicht zwingend erforderlich ist und deswegen nicht als echte Graph-Annotation gehandhabt wird. Die Fremdheit wird weitergehend für keinen weiteren Faktor oder Metrik benutzt und dient damit ausschließlich der Berechnung der vererbten Redundanzen.

Insgesamt ist damit die Höhe der Redundanz auch ein Wert, welcher nach seiner Berechnung keinerlei Aussage über die genutzten *Frames* mehr gibt. Da diese jedoch später keine Relevanz mehr besitzen und ein einzelner Wert für die Höhe der Redundanz besser zu handhaben ist, wurde diese Vorgehensweise bevorzugt. Eine weitere Eigenschaft sollte hier noch genannt werden, und zwar, dass viele Datenbestände mit Primär- und Fremdschlüsseln aufeinander referenzieren.

id	f_name	l_name	tel	adr	reg	hungry	notes
42	Cave	Johnson	1234 5678	ABCD	19.04.2011	×	todo

Quell-Datenbestand d_q „Clients“

id	→ c_id	fname	lname	state	city	street	number
1	42	Cave	Johnson	A	B	C	D

Ziel-Datenbestand d_z „Addresses“

Tabelle 5.2: Die Kopie von Daten über die *Frames*

Dadurch, dass diese tatsächlich auch eine Form der Redundanz darstellen, da einzelne Spalten zum Beispiel die `ID` einer anderen Tabelle als Referenz abspeichern, sollten diese Fremdschlüssel nicht mit in Betracht gezogen werden, da man hierbei davon ausgehen kann, dass diese Referenzen bewusst eingefügt und synchronisiert sind.

Annahme 9
Die gemanagten Fremd-Schlüssel (Referenzen) eines Datenbestandes sind immer im synchronisierten Zustand und stellen keine schädliche Redundanz dar. Zudem haben sie keinen Einfluss auf den Wertverlust durch unmanaged Redundancy.

Um die Berechnung des *Rd*-Wertes weiter zu verdeutlichen, sei hier das tabellarische Beispiel aus Abbildung 5.1 weiter oben nochmals herangezogen und in Tabelle 5.2 verfeinert. Mann kann hier drei Arten von kopierten Werten erkennen:

1. Die grünen *Frames* sind Referenzen (durch Fremd-Schlüssel organisiert und per → gekennzeichnet), die benötigt werden, um der Adresse weitere Informationen (hier der zugehörige Kunde) hinzuzufügen. Da dies aus technischer Sicht teilweise unvermeidbar ist, wird diese Art der Redundanz vernachlässigt. Sollten einmal alle Werte aus einer Zeile nur aus Referenzen bestehen, ist die Zeile faktisch Redundanz-frei, da bei echten Referenzen davon ausgegangen wird, dass diese immer synchronisiert sind, wie in der Annahme erläutert wurde.

2. Die blauen *Frames* sind direkt kopierte Werte und stellen keine echte Referenz dar. Bei diesen wurden einfach neue Spalten in d_z angelegt und Werte werden extern synchronisiert oder nicht. Dies stellt die einfachste Form der Redundanz dar. Wie man erkennen kann, können sich die Namen der Spalten durchaus unterscheiden, was das spätere Auffinden verhindern kann.

3. Die roten *Frames* sind ebenfalls redundante Werte, nur dass hier, neben anderem Namen, die Information auch in eine andere Form gebracht wurde. Dies ist die am schwierigsten zu findende Redundanz im System und kann erhebliche Fehler verursachen, wenn diese Redundanz nicht synchronisiert ist.

Nimmt man sich die oben definierte Metrik zur Hand, kann die *unmanaged* Datenredundanz zwischen beiden Datenbeständen bestimmen:

$$\overline{Rd} = \frac{cF_{d_q}}{F_{d_q}} \cdot \frac{cF_{d_z}}{F_{d_z}} = \frac{3}{8} \cdot \frac{6}{8} \approx 0.28$$

id	f_name	l_name	tel	→ adr_id	reg	hungry	notes
42	Cave	Johnson	1234 5678	1	19.04.2011	×	todo

Quell-Datenbestand d_q „Clients“

id	→ c_id	state	city	street	number
1	42	A	B	C	D

Ziel-Datenbestand d_z „Addresses“

Tabelle 5.3: Wie die Kopie (Referenzierung) von Daten eigentlich aussehen müsste

Und obwohl man erkennen kann, dass die neue Datenbank zu 75% nur Werte aus der Quelle kopiert, liegt das Verhältnis der Redundanz bei „nur“ 0.28 als Maß für die inhaltliche und strukturelle „Gleichheit“. Das funktioniert natürlich auch bei mehr als jeweils einer Zeile im Datenbestand, dann sind die Werte für cF und F entsprechend höher. Wie man das Beispiel ganz ohne Redundanz lösen kann, zeigt Tabelle 5.3 auf, in dem alle notwendigen Werte entweder nur einmalig vorkommen oder entsprechend referenziert sind. Damit ist $cF = 0$ und daraus ergibt sich $\overline{Rd} = 0$. In der Praxis würde das bedeuten, dass der Entwickler mehr Zeit in das Design der Datenbank und in deren verknüpfte SQL-Anfragen stecken müsste, was bei diesem Beispiel zwar noch harmlos sein mag, in der Praxis jedoch erhebliche Auswirkungen auf den Mehraufwand haben kann, wenn die Datenbanken mehrere hundert *Frames* besitzen und eine Information auf viele Datenbestände per Referenz gesplittet wird.

Falls dieses theoretische Konzept mit dem Redundanz-Verhältnis *Rd* der „Gleichheit“ in der Praxis sich als (teilweise) falsch herausstellen sollte, wovon zwar nicht ausgegangen wird, was aber dennoch durch beispielsweise empirische Messungen passieren kann, dann kann statt dem neuen *Rd*-Verhältnis auch das klassische Verständnis von Redundanz benutzt werden, welches aussagt, wie viel Prozent der Daten im Ziel kopiert sind, ohne Rücksichtnahme auf die Quelle der Kopien. Durch den Ersatz ändert sich in der Graph-Metrik nicht allzu viel, da auch hierbei nur Werte zwischen 0 und 1 entstehen würden, welche dann an die Pfeile annotiert werden. Natürlich können dadurch jedoch Änderungen an den Wert-Metriken entstehen.

Damit sind alle Arten der Redundanz im Graphen formal beschrieben und dahingehend erläutert, dass man bei Bedarf in dieser Darstellung des Problems durchaus einige variable Punkte nutzen kann, um die Graph-Metriken entsprechend anzupassen. Hierbei sei noch einmal kurz die Aufgliederung der drei Redundanz-Arten zu Übersichtszwecken gegeben.

1. Die *Standard-Redundanz* beschreibt die „Gleichheit“ der Inhalte und Struktur zweier Datenbestände und existiert im Graphen als durchgezogene, rote Kante mit einer einfachen *Rd*-Annotation. In Formeln wird diese als $\overline{Rd}$ notiert.

2. Die *synchronisierte Redundanz* ist eine hinreichend gesicherte Kopie von Daten in verschiedenen Datenbeständen und wird im Graphen als durchgezogene, grüne Kante dargestellt und besitzt ebenfalls eine einfache *Rd*-Annotation. In Formeln wird diese als *Rd* dargestellt.

3. Die *vererbte Redundanz* entsteht aus synchronisierten Redundanzen und verläuft im Graphen als gestrichelte, rote Kante immer parallel zur zugehörigen synchronisierten Redundanz. Sie besitzt einen berechneten *Rd*-Wert, welcher sich aus dem *Rd*-Wert der parallelen,

synchronisierten Redundanz und der schadhaften Fremdheit σ' des Zieldatenbestandes ergibt. In Formeln wird sie entweder explizit als $\widehat{Rd}$ dargestellt oder mit der Standard-Redundanz $\overline{Rd}$ zusammengefasst.

Im folgenden Abschnitt wird auf die Wert-Metriken eingegangen, welche im eigentlichen Sinne viel wichtiger sind als die Graph-Metriken, da diese den Wert des Systems, oder vielmehr dessen Wertverlust durch Mehraufwand, numerisch ermitteln sollten. Diese Wert-Metriken sind die Messbarkeit auf der Darstellung, welche die Graph-Metriken bereitstellen. Durch sie können die Eigenschaften in dem Graphen sinnvoll integriert werden und dabei ausreichend Platz für Variabilität bieten, diverse Anpassungen durchzuführen.

5.3.2 Wert-Metriken

Die Wert-Metriken sind dazu da, die Messbarkeit durchzuführen. Nachdem nun die Darstellung über den Graphen und dessen zugehörige Metriken beschrieben und definiert wurde, werden nun die eigentlichen Metriken dafür entwickelt, welche eine Beziehung zwischen den strukturellen Eigenschaften des Systems und den Wert des Systems schaffen, um die Theorie dahingehend zu bestätigen.

Wie bereits erläutert wurde, wird der Wert des System in dieser Arbeit über den Aufwand und nötige Investition für das System definiert, was als Grundlage für dem Wert (besonders der Agilität) dient. Ein System, welches hohe Investitionen benötigt, ist kausal weniger wert als ein System, welches bei exakt gleicher Funktionalität weniger Investition erfordert.

Definition 2
Der Aufwand und die Investition in ein System gibt Aussage über dessen Wert, indem „zusätzlich notwendiger Mehraufwand“ als Wertverlust des Systems angesehen werden kann, da dieser nur für Systemreparatur berechnet wird und keinen echten Mehrwert (Funktionalität) am System erzeugt.

Die Idee hinter allen Wert-Metriken ist, ein einzelnes System zu den beiden Zeitpunkten t und $t + \tau$ auf ihren Aufwand hin zu evaluieren, und anschließend miteinander vergleichen zu können. Der Aufwand, welcher in ein System gesteckt wird, wurde bereits weiter oben geschildert und wird über eine inhaltliche Größe beschrieben. Die *Function Points* geben einen guten Anhaltspunkt als Angabe über das Ausmaß an Investition, das in ein System gesteckt werden muss. Würde man eine gewisse Anzahl *FP* als Basis-Investition eines Systems setzen, könnte man über einen Unterschied an *zusätzlich notwendigem Aufwand* (= „Additional Required Investment“ *ARI*) den Mehraufwand ermitteln, der Aufschluss über zusätzliche *TtM* und *DevC* liefert. Dieser zusätzliche Aufwand *ARI* ist dann nur dadurch entstanden, dass *Technical Debt* und *Architecture Erosion* in dem System vorzufinden sind. Man muss dabei aber bedenken, dass *FP* immer noch eine Metrik für die inhaltliche Größe der Funktionalität darstellt. Damit ist es nicht möglich, zusätzlichen Aufwand zu ermitteln, da durch schlechte Architektur die Funktionalität des Systems niemals erweitert wird, was allerdings eine erhöhte Anzahl an *FP* vermuten lassen würde. Dazu wird in dieser Arbeit zwischen den tatsächlichen Basis-*Function Points* und den *effective Funktion Points* (kurz *eFP*) unterschieden, welche Aussagen darüber treffen, wie viel Mehraufwand betrieben werden muss. Die *eFP* sind an sich keine inhaltliche Größe mehr sondern vielmehr eine Aussage, wie viel tatsächlicher Aufwand in eine Komponente gesteckt werden muss, um diese fertigzustellen. Damit kann eine 40 *FP*-Komponente einen *eFP*-Wert von 42 besitzen, was die Funktionalität der Komponente nicht um 2 Punkte erweitert, sondern

besagt, dass für die Komplettierung der 40 *FP* ein Aufwand betrieben werden muss, der einer 42 *FP*-Komponente gleicht, obwohl die inhaltliche Größe sich nicht verändert und bei 40 *FP* bestehen bleibt. Damit stellen die *eFP* die Summe im Aufwand dar, welcher oben beschrieben wurde und sind dadurch geeignet, als Messwert für den Wertverlust zu dienen. Um nur einen finalen Wert darzustellen, beinhalten *eFP* nicht nur die Differenz-*FP*, sondern werden als

$$eFP = FP + ARI$$

verstanden. Stellt sich noch die Frage, womit der Wert des *ARI* ermittelt werden soll. Dazu sind die in diesem Kapitel vorgestellten Wert-Metriken benutzbar, welche als Resultat der Zähloperationen auf dem Graphen einen einzelnen numerischen Wert für *ARI* im Sinne der *eFP* ermitteln. Damit dies mit den normalen *FP* verglichen werden kann, muss der *ARI* pro Komponente im Graphen ausgewertet werden, was allerdings kein Problem darstellt, da die Menge an Komponenten endlich ist. Weil die Unterscheidung hierbei essenziell ist, sind die drei Werte noch einmal kurz zusammengefasst:

FP ist die feste inhaltliche Größe von Komponenten,

eFP ist der summierte, tatsächliche Aufwand bei der Entwicklung einer Komponente, gebildet aus $eFP = FP + ARI$,

ARI ist der zusätzlich erforderliche Aufwand für eine Komponente. Der *ARI*-Wert ist dabei das Resultat der Wert-Metriken und stellt keine Funktionalität dar.

Im Folgenden werden die drei Wert-Metriken vorgestellt, welche diverse Eigenschaften und Graph-Metriken nutzen, um geeignet kausal einen einzelnen numerischen Wert wiederzugeben. Nach Fenton *et al.* [16] sollten Metriken möglichst konsistent auf ihre Eigenschaften definiert werden. Dabei spielen vor allem folgende vier Werte des Öfteren eine Rolle:

1. *Nonnegativity* als die Eigenschaft, dass der Wert der Metrik nicht kleiner Null sein kann und immer ≥ 0 *eFP* berechnet wird,
2. *Null Value* als Eigenschaft, dass die Metrik einen neutralen Wert kennt, der sogenannte „Nullwert“, idealerweise die Metrik also 0 *eFP* für nicht-vorhandene schadhafte Redundanz berechnet,
3. *(Non)Symmetry* als Eigenschaft, dass die Reihenfolge der internen Elemente der Metrik (k)einen Einfluss auf das Resultat besitzt und die Reihenfolge von Iterationen (k)eine Rolle spielt, und
4. *Monotonicity* als Eigenschaft der Metrik, dass sich die berechneten *eFP* auch im Sinne der Gleichmäßigkeit verhalten und sich andere Inputs auch wie erwartend verhalten.

Auf die entsprechenden Eigenschaften der Metriken wird genauer bei der Zusammenstellung dieser eingegangen. Dabei deckt jede der drei Metriken einen Kosten-Bereich ab: *Adaptive*, *Corrective* und *Preventive Maintenance*, welche bereits in Kapitel 2.1.2 „Die Kosten eines Systems“ beschrieben wurden. Damit wäre im Endeffekt der Zusammenhang zwischen dem Architektur-Wert und der Datenredundanz als Systemeigenschaft gegeben. Die Vorgehensweise in diesem Kapitel wird der kausalen Kette an Argumentation folgen, wodurch die Metriken entstanden sind, um die einzelnen Teile der finalen Wert-Metrik zu erläutern. Dabei ist zu beachten, dass die hier entstehenden Metriken keine komplexen mathematischen Formeln darstellen sollen, sondern vielmehr als Zähl-Algorithmen angesehen werden, die auf dem Graphen ausgeführt werden

können. Um den Aufbau der einzelnen Metriken nachvollziehen zu können, werden neu hinzugefügte Teile der Metrik in jedem Schritt rot markiert und ein kleines Diagramm am Ende jeder Metrik gibt nochmal einen Überblick über die einzelnen vollzogenen Schritte. Ein Hinweis, welcher auf alle folgenden Metriken zutrifft, sollte hier noch einmal genannt werden: die schadhafte Redundanz (Standard und vererbt) wird in allen Formeln als $\overline{Rd}$ zusammengefasst. Denn obwohl die vererbte Redundanz explizit als $\widehat{Rd}$ dargestellt werden kann, betreffen die Zähloperationen immer alle schadhaften Redundanzen, weswegen es egal ist, ob der schadhafte *Rd*-Wert durch eine Annotation einer Standard-Redundanz oder einer berechneten *Rd*-Größe einer vererbten Redundanz herführt. Das bedeutet, dass in den Formeln eine Iteration über alle $\overline{Rd}$-Beziehungen auch alle $\widehat{Rd}$-Beziehungen erfasst. Allerdings müssen die entsprechenden $\widehat{Rd}$-Werte dennoch korrekt vorher erfasst werden.

Entsprechend den in Kapitel 3.4.1 „Software Produkt Attribute“ genannten Kategorien kann man die Wert-Metriken ebenfalls für das Attribut *Komplexität* einordnen, diesmal im Gegensatz zu den Graph-Metriken als *externes* Attribut, da bei den Wert-Metriken auch externe Werte außerhalb der Grundlage (dem Graphen) einspielen werden. Während das Attribut der Metrik zwischen Graph- und Wert-Metriken sich nur unerheblich ändert, ändert sich jedoch der Scope. War dieser bei den Graph-Metriken *Structural and complexity metrics*, ist der Scope bei den Wert-Metriken *Cost and effort estimation*. Hiermit ist gleichzeitig ein gutes Beispiel gegeben, wie sich eine Metrik aufgrund von Attribut und Scope ändern kann.

Adaptive Maintenance Die strukturellen Äußerungen im System durch geringe *Adaptive Maintenance* sind der prinzipielle Vorteil des Systems, neue Anforderungen an das System schnellstmöglich und kostengünstig umsetzen zu können. Eine neue Anforderung kann dabei durch neue Kundenwünsche, neue Technologien oder zwingende Erweiterungen gegeben sein. Soll etwas Neues in das System verbaut werden, müssen alte Komponenten in ihrer Funktionalität erweitert werden oder neue Komponenten hinzukommen. Dieses Verhalten wird im Graphen dadurch repräsentiert, dass bei neuen Anforderungen eine neue Komponente hinzugefügt wird. Soll nur eine bereits bestehende Komponente erweitert werden, wird diese neue Funktionalität entweder „abgesplittet“ und virtuell in eine neue Erweiterung eingeordnet (wenn diese unabhängig erscheint), oder aber die alte Komponente wird aktualisiert. Das ist deshalb möglich, weil in der gewählten Vorgehensweise bei jedem Systemzustand alle Komponenten und Beziehungen erneut ausgewertet werden, was sowohl alte als auch neue Elemente umfasst. Damit können durch die modellierte Systemevolution im Graphen verschiedene Vorgehensweisen modelliert werden, je nach Verhältnis zwischen „Neubau“ oder „Erweiterung“ der Komponenten im System.

Metrik-Idee

Die Idee dahinter ist, dass durch die Metrik der *Adaptive Maintenance* ein Wert gefunden wird, der die negativen Folgen durch und bei Änderungen am System widerspiegelt. Diese negativen Folgen ergeben sich zum Beispiel durch längere Baukosten für zukünftige Erweiterungen und höheren Wartungsaufwand. Insgesamt steht die Metrik dafür, was passiert, wenn ein Entwickler neue Funktionalität einbauen möchte, aber in eine veraltete Architektur zu integrieren hat. Dies bedeutet natürlich für ihn (je nach Verfall des Systems) längere Prozesse und damit die oben genannten negativen Folgen, sodass für eine 40 *FP*-Komponente ein Aufwand entsteht, der mehr oder minder deutlich die übliche Zeit für 40 *FP* übersteigt (zum Beispiel durch höheren Programmier- und Test-Aufwand und weniger Überblick durch mehr Komplexität).

Der Vorteil von einer neuen Anforderung, beziehungsweise der *Adaptive Maintenance*, ist, dass man recht genau die *FP* der neuen Komponente bestimmen kann. Die oben genannten vier Eigenschaften sollen dabei wie folgt gelten:

1. die *Nonnegativity* ist gegeben, sodass eine neue Komponente nicht negativen *ARI* aufweisen kann,

2. der *Null Value* ist dann gegeben, wenn das Resultat der Metrik (*ARI*) gleich Null ist, wenn auch die *unmanaged Redundancy* $\overline{Rd} = 0$ ist,

3. die *Symmetry* ist dadurch gegeben, dass es keine Rolle spielt, in welcher Reihenfolge verschiedene Elemente in die Metrik einfließen, und

4. die *Monotonicity* ist dadurch gegeben, dass eine höhere *unmanaged Redundancy* (der „Schaden“) einen höheren *ARI*-Wert verursacht als eine niedrigere *unmanaged Redundancy*, basierend auf der Redundanz *Rd*.

Diese vier Eigenschaften sollen sowohl auf diese Metrik zutreffen, als auch auf die folgenden Metriken, da diese Eigenschaften universell sind, weswegen sie später nicht noch einmal explizit aufgezählt werden. Wie bereits erwähnt, muss eine solche Metrik auf alle einzelnen Komponenten angewendet werden, um sinnvoll die Zähloperation auf Basis der *FP* ausführen zu können. Die entstehende Metrik soll widerspiegeln, was der tatsächliche Aufwand einer neuen Komponente (oder einer alten erweiterten Komponente) ist, wenn diese auf eine *unmanaged Redundancy* aufbaut. Sicherlich hängt der finale Mehraufwand auch von anderen Entscheidungen ab, aber hier wird nur um den Schaden der Datenredundanz betrachtet, weswegen alle anderen Einflüsse unbeachtet bleiben. Bleibt noch die Frage, welche Faktoren die *Adaptive Maintenance* in Bezug auf die Datenredundanz beeinflussen. Zum Einen ist es sicherlich die inhaltliche Größe der Komponente, um das Ausmaß zu erkennen. Ebenso unvermeidlich ist es, die Redundanz mit einzubeziehen, da man nur damit den Einfluss modellieren kann. Letztlich wäre noch die Einbeziehung der Nutzung des Datenbestandes von Vorteil, da dies Auswirkungen auf das Entwickeln der Komponente besitzt.

Dadurch, dass die neu hinzugefügten *FP* der neuen Komponente bekannt sind, kann direkt mit ihnen gearbeitet werden. Der erste Schritt in der kausalen Kette hin zur finalen Metrik für *Adaptive Maintenance* ist deshalb, über alle Komponenten zu iterieren und deren *FP* zu analysieren. Kennt man deren *FP*-Werte, kann man später weitere Faktoren einfließen lassen, um die relativen Bezüge herzustellen. Damit ergibt sich der erste Teil der ersten Wert-Metrik:

$$y_a = \sum_{i=1}^{m} FP_{c_i}$$

Das y_a stellt das Symbol für den *ARI*-Wert für die *Adaptive Maintenance* dar. Allerdings werden dadurch nur alle *FP*-Werte der Komponenten $c_1, c_2, \ldots, c_{m-1}, c_m$ durch $FP(c_1)+FP(c_2)+\ldots+FP(c_{m-1})+FP(c_m)$ aufsummiert, was im Grunde den Gesamt-Basis-Aufwand für das System beschreibt. Der Kürze und Übersichtlichkeit halber wird ab hier in den Metriken selbst ein Attribut mit Fußnote statt Funktionsklammer notiert, sodass $FP(c_i) = FP_{c_i}$ und $E(d_i) = E_{d_i}$ ist. Später wird das FP durch eine Funktion C ersetzt, welche immer noch die *FP* einer Komponente zählt, jedoch berücksichtigt, wie viel an der Komponente geändert wird. Weitere Ausführungen dazu folgen später. Um die Metrik zu verfeinern und eine sinnvolle Metrik für den Mehraufwand zu bestimmen, müssen verschiedene Faktoren aus dem Graphen gefunden werden,

welche zum einen eine Relation zu der inhaltlichen Größe der Komponente c_i aufweisen können, und zum anderen auch entsprechend ihres Einflusses eingearbeitet werden können.

Der nächste größere Faktor ist, wie oben beschrieben, die Redundanz selbst. Greift eine Komponente auf eine *unmanaged Redundancy* zu, ist laut Annahme ein Schaden existent, welcher als Faktor in die Metrik eingearbeitet werden muss, um den zusätzlichen Aufwand *ARI* in *eFP* zu bemessen. Dabei ist der Schaden auf jeden Fall durch die *Monotonicity*-Eigenschaft in eine Richtung vorgegeben, was bedeutet, dass eine Steigerung der Redundanz (beziehungsweise des *Rd*-Wertes) eine Steigerung des Wertverlustes herbeiführt. Nach theoretischen Überlegungen wurde entschieden, dass die *unmanaged Redundancy* hauptsächlich linear eingeht, da weder eine Abschwächung noch eine Steigerung des Anstieges vom Aufwand zu erwarten ist, wenn die Redundanz größer wird. Eine höhere *unmanaged Redundancy* bedeutet für den Entwickler, dass er auf Fehler stößt, die er sich (vorerst) nicht erklären kann, weshalb zum Beispiel falsche Werte im Zusammenspiel mit anderen Komponenten auftauchen. Da man als Entwickler der Komponente jedoch etappenweise arbeitet und des Öfteren die Möglichkeit hat, seinen Code zu testen und auszuprobieren, bedeutet eine höhere Redundanz einen konstanten Anstieg des Aufwandes.

$$y_a = \sum_{i=1}^{m}(FP_{c_i} \cdot \sum_{j=1}^{n}\sum_{k=1}^{n}(\overline{Rd}_{d_j d_k})) \quad \text{mit } \exists\varphi_{c_i d_j}$$

Dabei gilt das n als maximaler Index (und damit maximale Anzahl) von Datenbeständen und die *unmanaged Redundancy*'s $\overline{Rd}$ im System von Datenbestand d_j hin zu Datenbestand d_k. Die zwei Summen sind im Grunde nur für die Iteration über alle Datenbanken vorhanden, die erste für den Ausgangspunkt der $\overline{Rd}$, die Zweite für den Endpunkt der $\overline{Rd}$. Der Zusatz mit $\exists\varphi_{c_i d_j}$ ist vorerst notwendig, da ansonsten alle Redundanzen gezählt werden, aber eigentlich nur jene zählen sollen, die auch von der Komponente c_i ausgehen und bei einem Datenbestand d_j enden. Deswegen wird hier die Bedingung eingebracht, dass eine solche Nutz-Beziehung auch existiert.

Der letzte große Faktor, welcher in die Metrik für *Adaptive Maintenance* einfließen sollte, ist das Verhältnis der Nutzung des entsprechenden Datenbestands. Eine große Komponente kann beim Entwickeln durchaus viel weniger Aufwand bedeuten als eine kleine Komponente, wenn deren Nutzung und Zugriff auf die Datenbestände stark variieren. Der Umfang der Nutzung wurde bereits weiter oben in den Graph-Metriken als *CRUD*-Wert im Vektor φ beschrieben. Dieser Faktor wird für jeden Datenbestand extra berechnet und ist pro Komponente-Datenbestand-Beziehung individuell. Damit kommt man auf die folgende Ausprägung der Metrik:

$$y_a = \sum_{i=1}^{m}(FP_{c_i} \cdot \sum_{j=1}^{n}\sum_{k=1}^{n}(\overline{Rd}_{d_j d_k} \cdot \overrightarrow{\varphi}_{c_i d_j}))$$

Das $\exists\varphi_{c_i d_j}$ kann nun entfallen, da ein $\varphi_{c_i d_j}$ dies im Grunde voraussetzt und gleichbedeutend ist. Der nun eingearbeitete Vektor $\overrightarrow{\varphi}$ (in der Metrik formal mit dem Pfeil als Kennzeichnung für einen Vektor dargestellt) wird direkt 1:1 auf die *unmanaged Redundancy* multipliziert, da beides im direkten Verhältnis steht und pro Komponente und pro $\overline{Rd}_{d_j d_k}$-Beziehung hinzu gerechnet wird. Dabei wird der φ-Vektor mit seinen Werten $\varphi_C, \varphi_R, \varphi_U, \varphi_D$ jeweils mit $\overline{Rd}_{d_j d_k}$ multipliziert und aufaddiert:

$$\overline{Rd}_{d_j d_k} \cdot \overrightarrow{\varphi} = \overline{Rd}_{d_j d_k} \cdot \varphi_C + \ldots + \overline{Rd}_{d_j d_k} \cdot \varphi_D$$

Da jeder Vektor-Wert von φ eine Höhe zwischen 0 und 1 aufweisen kann und $\overline{Rd}$ ebenfalls maximal 1 aufweist, liegt die obere Grenze für den Wert momentan bei 4.

Damit sind alle größeren Einflüsse einbezogen, auch wenn bisher alles noch im Faktor-Verhältnis 1:1 gleichgesetzt ist. Die erste Relativierung der Metrik erfolgt in Form des Nutzungsverhaltens des Datenbestandes. Es spielt dabei eine große Rolle, ob die Entwicklung einer Komponente mit Schreibzugriff die Daten verändern muss oder nur lesend die Daten verarbeitet, sodass verschiedene Arten auch verschieden große Auswirkungen auf den Aufwand besitzen, der beim Entwickeln anfällt. Diese Gewichtung von Arten der Nutzung eines Datenbestandes kommt nun als weiterer Faktor hinzu, welcher eine Aussage über das liefert, wie wichtig und schwierig es wird, die Komponente zu entwickeln. Dieser Vektor wird dabei folgendermaßen definiert:

$$\overrightarrow{\varphi}' = \begin{pmatrix} 0.25 \\ 0.10 \\ 0.40 \\ 0.25 \end{pmatrix}$$

Wie man erkennen kann, ist dies eine Komponenten-unabhängige Konstante in der späteren Metrik. Der Vektor φ' ist ein Spezial-Faktor, welcher nur die Funktion hat, den originalen Vektor φ zu relativieren:

$$\overrightarrow{\varphi} \cdot \overrightarrow{\varphi}' = \varphi_C \cdot \varphi'_C + \varphi_R \cdot \varphi'_R + \varphi_U \cdot \varphi'_U + \varphi_D \cdot \varphi'_D = \varphi_C \cdot 0.25 + \ldots + \varphi_D \cdot 0.25$$

Dabei repräsentieren die Werte in φ' das Verhältnis der Schwierigkeiten der einzelnen Zugriffsarten von *CRUD* untereinander. Dadurch werden die unterschiedlichen Hürden beim Entwickeln der Komponente erfasst, welche anstehen, sollte man Daten „nur“ lesen oder doch erstellen, updaten und löschen. Da, wie oben beschrieben, jeder Wert aus φ einen Wert zwischen 0 und 1 besitzt, welcher mit einem Wert aus φ' multipliziert wird und φ' selbst in Summe 1 ergibt, ist die gesamte Summe $\overline{Rd}_{d_j d_k} \cdot \varphi \cdot \varphi'$ auf den Bereich $[0, 1]$ relativiert. Dass dieser Normalisierungsvektor φ' genau diese Werte besitzt, ist derzeit nur aus theoretischen Überlegungen begründet. Diese Werte werden in der Praxis sehr wahrscheinlich andere Kennzahlen aufweisen, jedoch das Prinzip der unterschiedlichen Gewichtung der Nutzungsarten repräsentieren können, weswegen die genauen Werte im Vektor nach empirischen Studien durchaus änderbar sind. Die einzige Bedingung dabei bleibt die Summe von 1 in φ'. Insgesamt ergibt sich nun folgende Metrik:

$$y_a = \sum_{i=1}^{m} (FP_{c_i} \cdot \sum_{j=1}^{n} \sum_{k=1}^{n} (\overline{Rd}_{d_j d_k} \cdot \overrightarrow{\varphi}_{c_i d_j} \cdot \overrightarrow{\varphi}'))$$

Der zweite Faktor zur Relativierung ist ein *Standard-Kronecker-Delta* [56]. Dieses ist dafür verantwortlich, eine bestimmte Situation auszuschließen oder einzubeziehen. Für die *Adaptive Maintenance* wurde es für die Situation eines Shortcuts bei der Entwicklung gewählt. Diese Situation geht auf die reale Praxis zurück und beschreibt das bewusste Einführen von *Technical Debt* als Einsparung von Zeit und Geld durch den Shortcut. Ein Entwickler, welcher einen Shortcut nutzt, macht dies vermutlich, um Zeit für eine saubere Implementierung zu sparen. Dieser lokale Boni an Zeitgewinn muss auch in der Metrik widergespiegelt werden, da hierbei eine negative Anrechnung an Zeit und Geld nicht sinnvoll wäre. Natürlich wird sich der Shortcut bei zukünftigen Entwicklungen negativ auswirken, spätestens wenn andere Komponenten auf die *Technical Debt* angewiesen sind, aber für den Zeitpunkt der Erstellung des Shortcuts ergibt sich kein Nachteil bezüglich des *ARI*, beziehungsweise des Mehraufwandes durch *eFP*. Dahingehend wird die Situation, dass ein Entwickler einen Shortcut nutzt und eine Datenbank mit *unmanaged Redundancy* erstellt, dahingehend gewertet, dass für den Zeitpunkt der Entwicklung kein Malus

anfällt. Dafür wird das *Standard-Kronecker-Delta* benutzt, um den Fall zu negieren, dass die *unmanaged Redundancy* zeitgleich mit der Entwicklung der Komponente entsteht:

$$\delta_{c_i d_j} = \begin{cases} 1, & \text{wenn } c_i \in \mathbb{E}_x^+ \text{ und } d_j \notin \mathbb{E}_x^+ \text{ mit } x \in \mathbb{N}, \\ 0, & \text{sonst.} \end{cases}$$

Diese Fallunterscheidung besagt, dass der Faktor $\delta_{c_i d_j}$ genau dann 1 ist, wenn eine Komponente c_i zu einer Erweiterung $\mathbb{E}_x^+$ gehört, zu der nicht gleichzeitig auch der entsprechende genutzte Datenbestand d_j gehört, zu dem die Komponente c_i eine Nutzung im Sinne φ vornimmt. Damit ist ausgeschlossen, dass ein Mehraufwand durch *unmanaged Redundancy* angerechnet wird, wenn der Entwickler eigentlich Zeit durch einen Shortcut einspart. Damit ergibt sich mit dem Einfügen von $\delta_{c_i d_j}$ und Umstellen der Faktoren nun folgende Metrik:

$$y_a = \sum_{i=1}^{m} (FP_{c_i} \cdot \sum_{j=1}^{n} (\delta_{c_i d_j} \cdot \overrightarrow{\varphi}_{c_i d_j} \cdot \overrightarrow{\varphi}' \cdot \sum_{k=1}^{n} \overline{Rd}_{d_j d_k}))$$

$$\delta_{c_i d_j} = \begin{cases} 1, & \text{wenn } c_i \in \mathbb{E}_x^+ \text{ und } d_j \notin \mathbb{E}_x^+ \text{ mit } x \in \mathbb{N}, \\ 0, & \text{sonst.} \end{cases}$$

Als vorletzter Schritt wird nun das initiale FP_{c_i} wie angekündigt in die Funktion $C(c_i)$ übersetzt, welche die Änderungsrate einer Komponente mit einbeziehen soll. Dies ist sinnvoll, da eine 80-*FP*-Komponente durchaus viel Mehraufwand erzeugen kann, dieser Mehraufwand jedoch theoretisch kleiner wird, wenn man die 80 *FP* Stück für Stück aufbaut. Solange man kleinere Schritte an Änderungen (als Erweiterungen) einer Komponente durchführt, ist der Einfluss schädlicher Strukturen kleiner. Wenn man nur eine Funktion hinzufügt, dann ist die davon beeinflusste Struktur relativ klein, als wenn man viele Abhängigkeiten einer großen Erweiterung vornimmt. Im Grunde wird damit nur beachtet, inwieweit der Inhalt einer Komponente im Laufe aller Erweiterungen als Schätzung für den Mehraufwand dienen kann. Dafür wird die *Content Rate of Change*, kurz *C* betrachtet, welche einfach nur die gesamten *FP* einer Komponente durch die Anzahl aller der Erweiterungen teilt, in der auch die betrachtete Komponente beteiligt war.

$$C(c_i) = \frac{FP_{c_i}}{\sum \mathbb{E}_y^+} \text{ mit } c_i \in \mathbb{E}_y^+$$

Wurde also die 80-*FP* Komponente im Laufe von 5 Erweiterungen gebildet, wird für die *Adaptive Maintenance* in der Metrik keine 80 *FP* genutzt, sondern die durchschnittliche Anzahl an 16 „Änderungs“-*FP* durch $C(c_i) = \frac{80}{5} = 16$. Doch was bedeutet das, dass die Komponente keine 80 sondern nur 16 *FP* in die Metrik für den Mehraufwand „einbringt“? Das bedeutet, das für den Mehraufwand nur ein Bruchteil der tatsächlichen inhaltlichen Größe für die *Adaptive Maintenance* zur Geltung kommt. Beide Ansätze (Rechnung per *FP* oder $C(FP)$) haben ihre Gründe, verwendet zu werden. Der Grund, warum für *C* entschieden wurde, ist, dass der Graph, so wie er derzeit vorliegt, keine Relativierung bei den zeitlichen Abläufen zwischen Komponenten und Redundanzen schafft. Eine Komponente kann 90% der „Zeit“ ohne Probleme im System existieren, bis eine neu hinzugefügte Nutzbeziehung zu einem schadhaften Datenbestand aufgebaut wird und die Komponente Einbußen in Form von zusätzlichen *eFP* bekommt. Deswegen muss ein anderer Faktor verwendet werden, der in Abhängigkeit der Erweiterungen einer Komponente den Einfluss aller *FP* relativiert. Dies geschieht durch die *C*-Funktion, welche mit ihrem Wertebereich zwischen $0 < C(FP_{c_i}) < FP_{c_i}$ definiert ist. Da damit die Erweiterungen einbezogen werden, in denen die Komponente beteiligt ist, wird die inhaltliche Größe bei der

Adaptive Maintenance auf den Schnitt aller „Änderungs"-*FP* der Komponente beschränkt. Die *C*-Funktion kann jederzeit wieder durch die „reinen" *FP* in der Metrik ersetzt werden, allerdings müssen dann eventuell andere Faktoren daran angepasst werden. Es sei angemerkt, dass eine Komponente im Graphen als Teil mehrfacher Erweiterungen noch entsprechend markiert werden muss, da eine Komponente durchaus mehreren Erweiterungen angehören kann, je nachdem, ob und wie sie erweitert wird. Sobald eine neue Komponente entsteht und nie wieder erweitert wird, ist $C(c_i) = FP(c_i)$. Damit steht die formale Metrik als Zwischenstand mit:

$$y_a = \sum_{i=1}^{m}(C(c_i) \cdot \sum_{j=1}^{n}(\delta_{c_i d_j} \cdot \overrightarrow{\varphi}_{c_i d_j} \cdot \overrightarrow{\varphi}' \cdot \sum_{k=1}^{n} \overline{Rd}_{d_j d_k}))$$

$$\delta_{c_i d_j} = \begin{cases} 1, & \text{wenn } c_i \in \mathbb{E}_x^+ \text{ und } d_j \notin \mathbb{E}_x^+ \text{ mit } x \in \mathbb{N}, \\ 0, & \text{sonst.} \end{cases}$$

$$C(c_i) = \frac{FP_{c_i}}{\sum \mathbb{E}_y^+} \text{ mit } c_i \in \mathbb{E}_y^+$$

Zum Schluss wird noch eine letzte Variable eingeführt, welche die theoretischen Überlegungen in der Praxis sinnvoll darstellen lassen. Das betrifft in erster Linie das Unternehmen oder das Projekt an sich, denn viele Einflüsse auf den Mehraufwand hängen auch an dem mehr oder weniger geschulten Team, an der verfügbaren Technik oder den vorherrschenden Entwicklungsprozessen. Dies sind alles Dinge, die nicht gut kausal erfassbar sind aber dennoch einen hohen Stellenwert für die notwendige Investition in das System besitzen. Ein Beispiel hierfür ist der Entwickler *A* im Vergleich zum Senior Entwickler *B*, welcher einen *FP* schneller umsetzen kann als der Entwickler *A*. Ein zweites Beispiel ist der vorherrschende Anforderungsanalyse-Prozess im Unternehmen, wie schnell neue Anforderungen in neuen *FP* resultieren. All das sind Werte, die nur schwer erfasst werden können, jedoch bedacht werden müssen. Dafür wird in dieser Arbeit die Variable q eingeführt, genannt „Unternehmens-Variable", welche entsprechend belegt werden muss. Diese kann niemals negativ sein, jedoch ist die Obergrenze bislang unbekannt. Es müsste in der Praxis festgestellt werden, welche Werte q annehmen kann, damit das Resultat der Wert-Metrik auch realistisch ist. Deswegen wird in den drei Wert-Metriken der Wert q eingefügt, welcher zwar in dieser Arbeit permanent mit 1 belegt wird, in der Praxis aber sehr vermutlich andere Werte aufweisen würde, was jedoch das Verhältnis der einzelnen Faktoren in der Metrik untereinander nicht stört sondern lediglich das Ergebnis in der Höhe relativiert.

Fügt man diese neue Unternehmens-Variable q_a ein (a für den Teil der Unternehmens-Variable speziell für den Kostenbereich der *Adaptive Maintenance*), ergibt sich die folgende finale Metrik als Zähloperation für *Adaptive Maintenance*:

$$y_a = \sum_{i=1}^{m}(C(c_i) \cdot \sum_{j=1}^{n}(\delta_{c_i d_j} \cdot \overrightarrow{\varphi}_{c_i d_j} \cdot \overrightarrow{\varphi}' \cdot \sum_{k=1}^{n} \overline{Rd}_{d_j d_k})) \cdot q_a$$

$$\delta_{c_i d_j} = \begin{cases} 1, & \text{wenn } c_i \in \mathbb{E}_x^+ \text{ und } d_j \notin \mathbb{E}_x^+ \text{ mit } x \in \mathbb{N}, \\ 0, & \text{sonst.} \end{cases}$$

$$C(c_i) = \frac{FP_{c_i}}{\sum \mathbb{E}_y^+} \text{ mit } c_i \in \mathbb{E}_y^+$$

Diese Metrik verhält sich nun theoretisch wie gewünscht und sollte den Mehraufwand in *ARI* darstellen. Ergibt sich aus der Metrik ein Resultat von $y_a = 0$, dann gilt $eFP(\mathbb{E}) = FP(\mathbb{E})$ und das System ist frei von *ARI* bezüglich der *Adaptive Maintenance*. Ist jedoch $y_a > 0$, ergibt sich

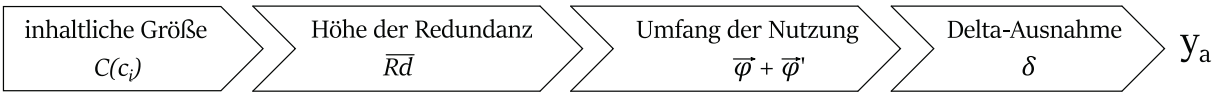


Abbildung 5.15: Die Zusammensetzung der *Adaptive Maintenance*

$eFP(\mathbb{E}) = FP(\mathbb{E}) + y_a$ als vorhandener Mehraufwand. Alle für *Adaptive Maintenance* relevanten Eigenschaften sollten nun darin vorkommen und damit einen ersten kleinen kausalen Zusammenhang geben können. Ebenso ist eine erste Quantifizierung durch die *eFP* gegeben. Weiterhin ist diese Metrik an diversen Punkten an die Praxis anpassbar, sollte sich etwas als zu realitätsfern herausstellen. Auch ist es möglich, weitere Faktoren in die Zähloperation einzubauen, sollten sie sich als relevant herausstellen.

In der Abbildung 5.15 sind nochmal alle großen Einflussfaktoren aufgezeigt, aus denen sich die Metrik zusammensetzt. Insgesamt kann man hierbei erkennen, dass es hauptsächlich um die erweiterte Komponente geht (als $C(c_i)$), welche mit den Metriken der Redundanz und dem Nutzumfang kombiniert wird. Die Delta-Ausnahme regelt schlussendlich die „negierten, temporären Verschlechterungen“ durch Shortcuts.

Corrective Maintenance Die *Corrective Maintenance* ist, im Vergleich zur *Adaptive Maintenance*, als etwas schwieriger anzusehen. Dies liegt hauptsächlich in der Natur der *Corrective Maintenance*, da, ganz im Gegensatz zu der *Adaptive Maintenance*, hierbei der Umfang des zusätzlich notwendigen Aufwandes *ARI* nicht durch eine neue Komponente mit festen *FP* bestimmbar ist. Gerade deswegen ist es nun wichtig, geeignete Schlüsse zu ziehen, um eine logische Erklärung für das Entstehen der Kosten für *Corrective Maintenance* zu finden. Genau das wird die folgende Metrik versuchen, indem gewisse Faktoren genutzt werden, um eine Art „Fehlerwahrscheinlichkeit“ zu finden, natürlich immer in Abhängigkeit der Datenredundanz, um die Kerntheorie nicht zu verfehlen.

Metrik-Idee

Die Intention hinter dieser Metrik ist, spätere Fehlerbeseitigungen und Fehlverhalten des Systems zu modellieren. Der ARI, welcher das Resultat dieser Metrik ist, stellt faktisch den Mehraufwand dar, den man später als Entwickler hat, durch neue Anforderungen alte *Technical Debt* zu entfernen. Das Problem liegt dann darin, die *unmanaged* Datenredundanz aufzufinden, eventuelle Korrekturen vorzunehmen und davon abhängige Komponenten möglichst fehlerfrei zu belassen. Ganz im Gegensatz zu der *Preventive Maintenance* geht man bei der *Corrective Maintenance* nicht davon aus, dass die *Technical Debt* (hier also die *unmanaged* Datenredundanz) gefixt wird, sondern „nur“ ein einfacher Weg gesucht wird, die Ursache zu finden, um dieses eine spezielle Problem zu lösen.

Im Grunde gibt es bereits viele statistische Modelle und Metriken für die Berechnung von der Stabilität, Zuverlässigkeit und Anfälligkeit von Software [16], jedoch bedienen sich viele derer an Dingen im System, die unabhängig der Datenredundanz betrachtet werden, wie zum Beispiel die Zeiträume zwischen verschiedenen Fehlern und die Perfektion von Fixes im Code. Obwohl alle Modelle ihre Daseinsberechtigung haben, und obwohl diese Modelle die tatsächlichen Wahrscheinlichkeiten und Kosten für die *Corrective Maintenance* besser einschätzen können als das Modell in dieser Arbeit, ist es doch eine komplett andere Intention, die diese Arbeit

von den bisherigen Modellen unterscheidet: der kausale und quantifizierte Zusammenhang von Datenredundanz, dem Wert des Systems und der *Corrective Maintenance*. So gesehen kann man die entstehende Metrik dieser Arbeit als Teil des Ganzen betrachten.

Die Idee bei dieser Metrik ist, darzustellen, inwieweit sich eine *unmanaged Redundancy* im System fortpflanzt und aufgrund verschiedener Nutzungen sich fehlerhafte Daten in Fehlverhalten im System äußern. Dabei wird absichtlich vernachlässigt, welche negativen Folgen wie Reputationsverlust, Strafzahlungen und andere Dinge auf das Unternehmen zukommen, sondern hierbei geht es lediglich um den Aufwand zum Finden und Fixen des Fehlers sowie der Verlust durch Ausfall einer Funktionalität. Auch wird hierbei wieder im Kern eine Zähloperation auf Komponenten entstehen, der zeitliche Aspekt wird durch die Erweiterungen im System realisiert. Dadurch wird immer das Gesamtsystem betrachtet und der Zeitfaktor spielt nur für den Zufallsfaktor der Fehlerquellen eine Rolle. Wie auch in *Adaptive Maintenance* sind die dort aufgelisteten vier Eigenschaften einer Metrik hier zutreffend.

Das Wichtige bei dieser Wert-Metrik ist, dass die Variabilität der Fehler durch *unmanaged* Datenredundanz eingearbeitet wird. Da das in der Darstellung definierte Modell eines Graphen deterministisch ist, gibt es keinen echten Ansatz eine Zufälligkeit einzuarbeiten, was dazu führt, dass diese auf anderem Wege eingearbeitet werden muss. Ebenso spielt für die *Corrective Maintenance* die Datenredundanz eine Rolle, welche den Wert in eine Richtung lenkt und die Orientierung vorgibt. Die inhaltliche Größe von Komponenten muss ebenfalls vorkommen, damit eine sinnvolle Berechnung von *eFP* möglich ist. Anders als in *Adaptive Maintenance* spielt bei der Metrik für *Corrective Maintenance* auch die Nutzung des Quell-Datenbestandes eine Rolle und nicht nur das φ zum Ziel-Datenbestand, da die Nutzung der Quelle eine Aussage darüber liefert, inwieweit sich die *unmanaged Redundancy* tatsächlich als Fehler überträgt. Hierbei wird darauf geachtet, nicht die eigentliche inhaltliche Größe der beiden relevanten Datenbestände zu betrachten, sondern tatsächlich nur deren Integration in das System durch einen Faktor, der angibt, von wie viel *FP* der Datenbestand „benutzt“ wird. Diese Unterscheidung ist deshalb wichtig, da auch sehr kleine Datenbestände (zum Beispiel zwei hart codierte *unmanaged* redundante Informationen) das Potenzial haben, größere Fehler hervorzurufen. Ein Einbauen eines Faktors der inhaltlichen Größe würde diesen Umstand nur unnötig abschwächen. Zuletzt wird es als sinnvoll erachtet, den Gesamtbezug des Fehlers zum System noch einzubeziehen, da man immer beachten muss, inwieweit die Ursache des Fehlers auch „getriggert“ wird. In einem System, wo die meisten Komponenten niemals etwas mit der *unmanaged* Datenredundanz zu tun haben, ist es unwahrscheinlicher, dass ein Fehler ausgelöst wird, als in einem System, in dem fast alles mit dem entsprechenden Datenbestand zu tun hat.

Der erste Schritt gleicht dem der *Adaptive Maintenance*, dem Iterieren über alle Komponenten:

$$y_c = \sum_{i=1}^{m} FP_{c_i}$$

Das hierbei definierte y_c wird später das Resultat der Metrik für *Corrective Maintenance*, den *ARI*, darstellen. Wie auch oben ist im ersten Schritt die Summe der inhaltlichen Größe aller Komponenten $FP(c_i)$ mit $FP(c_1) + \ldots + FP(c_m)$ gegeben, wobei m den maximalen Index und damit die maximale Anzahl an Komponenten darstellt. Die *FP* zu nutzen bedeutet hier, dass diese den Aufwand bestimmen werden. Anders als bei *Adaptive Maintenance* ist die inhaltliche Größe bei *Corrective Maintenance* dazu bestimmt, pro Komponente den potenziellen Aufwand für das Fixen der Fehler zu bestimmen. Das bedeutet, das die komplette Spannweite der enthaltenen *FP*, von 0 bis $FP(c_i)$, davon betroffen sein kann, je nachdem, welcher Fehler auftritt. Das Problem ist, dass man nicht voraussagen kann, welcher Fehler auftritt und wie viel der Funktio-

nalität der Komponente dieser betrifft. Das einzig Bekannte ist die Betrachtungsweise, dass die Fehler durch *unmanaged* Datenredundanz entstehen sollten. Es wurde im Rahmen dieser Arbeit entschieden, die inhaltliche Größe wieder linear einfließen zu lassen, da eine andere Formulierung nicht zielführend wäre.

Der nächste Faktor beschreibt die Zufallsgröße, mit der ein Fehler auftritt und welchen Umfang er besitzt, im Idealfall spezifiziert auf das System als Fehler-Modell. Diese Größe ist im Grunde eine eigene mathematische Funktion, welche je nach Kontext angepasst werden muss. Diese Funktion $r(\lambda)$ beschreibt eine Funktion oder ein Modell r in Abhängigkeit von einem Wert oder einer Metrik λ. Wie genau r und λ aussehen, ist eine Sache der empirischen Ermittlung, jedoch nicht einfach wie andere Faktoren vernachlässigbar. Das Problem besteht darin, dass ein Fehler auf unterschiedlichste Arten Auswirkungen haben kann, was im vornherein keine Leichtigkeit ist, einzuschätzen. Es gilt also, für $r(\lambda)$ ein geeignetes (minimalistisches) Modell des Fehlerverhaltens zu finden. Aufgrund der fehlenden empirischen Datenerhebung wurde beides so angelegt, dass ihre Semantik in dieser Arbeit dem Prozess des Fehlverhaltens wenigstens nahe kommt. Dabei werden r und λ wie folgt in dieser Arbeit belegt:

$$\lambda \sim [c_i]$$

$$r(\lambda) = r(\lambda|c_i) = \frac{y-x}{y} \text{ mit } y = MaxIdx(\mathbb{E}) \text{ und } c_i \in \mathbb{E}_x^+; \text{ wenn } y = 0 \text{ dann } r(\lambda) = 0$$

Man kann erkennen, das λ als „Kern“ des Fehler-Modells eine Komponente c_i nutzt (nicht zu verwechseln mit der Gleichsetzung $\lambda = c_i$, da hier nur „im Sinne“ zugewiesen wird, sozusagen als Grundbaustein der r-Funktion). Das ist einerseits sinnvoll, da die Kostenberechnung durch die Komponenten bestimmt werden soll, da diese die inhaltliche Größe vorgeben sollen. Andererseits ist es notwendig, da das Graph-Modell als Darstellung sehr minimalistisch gehalten ist (um die bestmögliche Kausalität zu erhalten), was dazu führt, dass sich keine großen Alternativen dazu anbieten würden. In zukünftigen Arbeiten wäre es möglich, den Graphen zu erweitern, um mit λ und r ein komplexeres Fehlermodell anzugeben.

Nachdem λ spezifiziert ist, gilt es, r zu definieren, wie es in der obigen Formel abgebildet ist. Dieses bildet ein Verhältnis aus $y - x$ und y, wobei das y den maximalen Erweiterungsindex im System $MaxIdx(\mathbb{E})$ (und damit der jüngsten Erweiterung) darstellt, und x der Index der Erweiterung ist, dem die zu untersuchende Komponente c_i angehört. Das bedeutet beispielsweise, wenn c_i aus $\mathbb{E}_4^+$ stammt und das System bereits bei $\mathbb{E}_{15}^+$ angelangt ist, dass

$$r(\lambda|c_i) = \frac{15-4}{15} = \frac{11}{15} \approx 0.73$$

ergibt. Sobald eine neue Erweiterung an das System angebaut wird (das heißt, man befindet sich im Zustand $\mathbb{E}_{16}^+$), erhöht sich der entsprechende Wert $r(\lambda|c_i)$ auf

$$r(\lambda|c_i) = \frac{16-4}{16} = \frac{12}{16} = 0.75$$

und tendiert damit mit jeder neuen Erweiterung gegen 1. Dieser Faktor dient damit als Relativierung (im Wertebereich $0 \leq r < 1$) der anderen Faktoren in der späteren finalen Metrik. Das Verhalten von $r(\lambda)$ stellt die vereinfachte Situation in der Praxis dar, inwieweit ein Fehler sich im Laufe der Zeit bemerkbar macht, neue Fehler aufkommen und damit unbeachtete *unmanaged Redundancy* sich auswirkt. So startet der Wert $r(\lambda|c_i)$ für eine Komponente bei 0 (wenn $y = x$) und verläuft kontinuierlich gegen 1 (wenn $y > x$) als obere Asymptote. Begründet wird dies

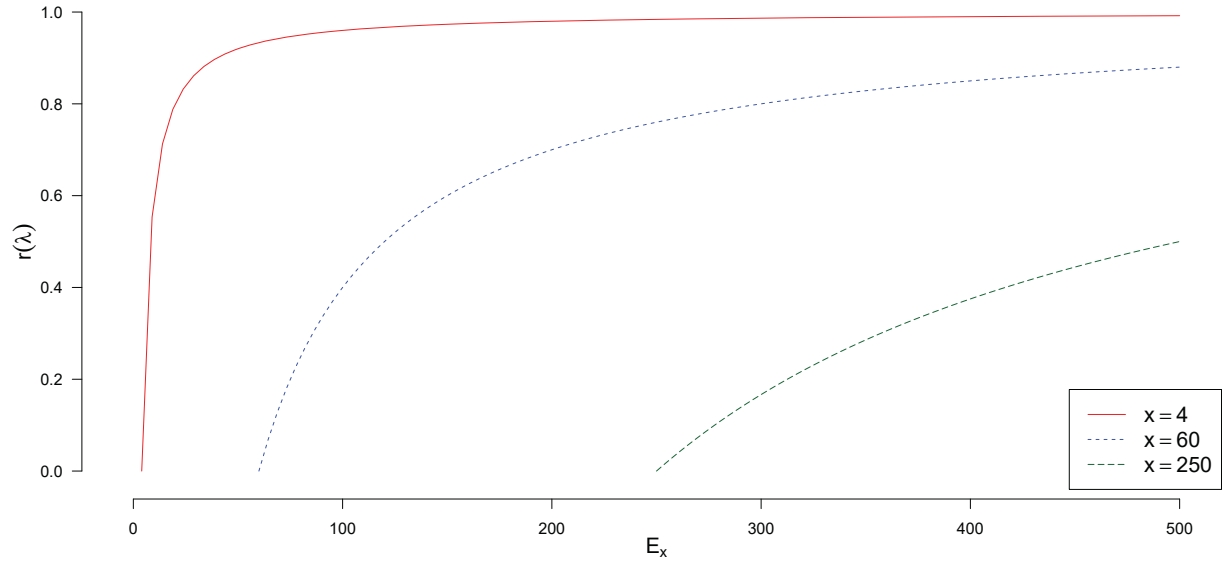


Abbildung 5.16: Der Verlauf der gewählten Funktion für $r(\lambda)$ von $\mathbb{E}_0^+$ bis $\mathbb{E}_{500}^+$

durch die Vorstellung, dass eine Komponente am System, welche neu hinzugefügt wird, bereits getestet wurde (eventuell ebenfalls mit Mehraufwand durch *Adaptive Maintenance*), was komplette neue Fehler eher ausschließt. Sobald neue Komponenten an das System angebaut werden, unabhängig deren Größe oder Einfluss, kann man theoretisch davon ausgehen, dass der einst eingebaute Fehler durch *unmanaged Redundancy* sich „entfaltet“ und seine Wirkung in Form von Fehlverhalten zeigt. Damit folgt die Begründung im Sinne der Begründung von $\sqrt{\overline{Rd}}$ weiter oben.

Der Verlauf dieser Funktion ist für eine Komponente c_i von $\mathbb{E}_0^+$ bis hin zum Zustand $\mathbb{E}_{500}^+$ einmal in Abbildung 5.16 dargestellt, wo sich sehr gut die Tendenz zu $r(\lambda|c_i) = 1$ für verschiedene Werte für x erkennen lässt. Sollte einmal $y = 0$ sein, ist diese Funktion mathematisch nicht definiert und damit normal erst ab $y = 1$ benutzbar. Sollte dennoch $y = 0$ anliegen, dann ist das gesamte Resultat hierbei auf 0 definiert. Zudem kann es nicht vorkommen, dass einmal $y < x$ ist, da der Index der betrachteten Erweiterung niemals größer sein kann als der Index der maximalen Erweiterung, weswegen die in der Abbildung gezeigten Kurven auch erst bei $x = 4, 60$ und 250 beginnen. Für größere Werte von x bedeutet diese Funktion, dass $r(\lambda|c_i)$ langsamer gegen 1 tendiert als bei kleinen Werten von x. Das kommt daher, dass zum Beispiel $\frac{y-60}{y}$ mit $y \geq 60$ einfach länger benötigt, um einen Bruch zu erzeugen, welcher $r(\lambda|c_i) = 1$ nahe kommt, als $\frac{y-4}{y}$, ebenfalls zu sehen in Abbildung 5.16. Dies ist der primitiven Natur der Funktion geschuldet. Die Funktion $r(\lambda)$ kann nach empirischen Ermittlungen angepasst werden, je nachdem, wie es sich am sinnvollsten in der Praxis herausstellt. Durch eine geeignete Wahl von $r(\lambda)$ kann die Metrik sehr realitätsgetreu die *eFP* des *ARI* ermessen, jedoch kann auch analog durch eine falsche Wahl von $r(\lambda)$ die Metrik ihren Sinn größtenteils verlieren. Deswegen sollte auf einer durchdachten Funktion aufgebaut werden. Die hier gewählte Funktion für $r(\lambda)$ scheint zumindest theoretisch geeignet, was allerdings keineswegs die perfekte Funktion darstellt um die Praxis zu simulieren.

Insgesamt ergibt sich damit der derzeitige Stand der Metrik durch:

$$y_c = \sum_{i=1}^{m} (FP_{c_i} \cdot r(\lambda|c_i))$$

Der nächste große Einflussfaktor für die *Corrective Maintenance* ist die Redundanz, welche in jeder Metrik vorkommen soll, da diese den Einfluss der Metrik darstellt. Die Redundanz wird wie auch bei der *Adaptive Maintenance* über zwei Summen aufgezählt, sodass alle nötigen Pfeile im Graphen erfasst werden. Der Unterschied ist hierbei, dass erstmalig eine Relativierung der Redundanz stattfindet, um geeignet die Praxis zu simulieren. Dazu wird die Redundanz $\overline{Rd}$ mit einer Wurzel versehen $\sqrt{\overline{Rd}}$, um darzustellen, wie sich höhere Redundanz genau auswirkt. Theoretisch gesehen ist eine Wurzel-Funktion geeignet, da diese am Anfang bei kleinen Redundanzen stark ansteigt und später bei höher werdenden Redundanzen weniger Anstieg besitzt. Das sollte zumindest die Basis korrekt darstellen, dass, wenn Fehler im Programm auftauchen, diese recht schnell komplex werden können, aber dann auch in ihrer Komplexität nach oben hin eingeschränkt sind. Ein Beispiel sei hierfür das Verschicken falscher Rechnungen aufgrund falschen Namens. Dies hängt dann zum Beispiel mit der *unmanaged Redundancy* einer Komponente zusammen, welche den „echten" Namen nicht synchronisieren kann. Das bedeutet, dass ein Entwickler nicht nur die Komponente zum Verschicken der Rechnungen und der Namenseingabe kontrollieren muss, sondern alle Komponenten auf dem Weg der Namensverarbeitung. Dies lässt den Aufwand vorerst schnell ansteigen. Davon abgesehen ist irgendwann bekannt, um welche Komponenten es sich handeln könnte und man kann davon ausgehen, dass gewisse andere Komponenten nicht betroffen sind, was schlussendlich die Abnahme des Anstiegs berechtigt, da schließlich irgendwann eine obere Grenze an Dingen erreicht wird, welche tatsächlich gesucht und gefixt werden müssen. Hierbei sei noch einmal darauf hingewiesen, dass die *Corrective Maintenance* nicht das Fixen der *unmanaged* Datenredundanz vornimmt, sondern nur den spezifischen Fehler korrigiert, mitunter durch neue Shortcuts (durchaus als „Quick'n'Dirty" bezeichnet). Damit ergibt sich der folgende Stand der Metrik:

$$y_c = \sum_{i=1}^{m} (FP_{c_i} \cdot r(\lambda|c_i) \cdot \sum_{j=1}^{n} \sum_{k=1}^{n} (\sqrt{\overline{Rd}_{d_j d_k}})) \quad \text{mit } \exists \varphi_{c_i d_j}$$

Bedingung hierbei ist, dass ein $\varphi_{c_i d_j}$ existieren muss, damit die Metrik auch ihren Sinn beibehält. Dass das $\varphi_{c_i d_j}$ hierbei nicht direkt hinein berechnet wird, liegt daran, dass der Umfang der Nutzung im nächsten Faktor bereits eingebaut ist und damit doppelt vertreten wäre.

Dieser nächste Faktor ist eine Kombination aus Nutzungsverhalten und Relativierung. Er stellt dar, inwieweit eine Datenbank tatsächlich benutzt wird. Dies wird genutzt, um grundlegend die Eigenschaft zu simulieren, wie ein Fehler sich im System fortpflanzt. Ein Datenbestand d_i, welcher durch *unmanaged Redundancy* aus d_j kopiert, hat damit nicht automatisch Mehrkosten für die Komponente, wie bei *Adaptive Maintenance*. Vielmehr treten die Fehler dann bei der Ausführung des Systems auf. Dazu wird das Verhältnis der *Nutzungsrate* der beiden Datenbestände d_i und d_j zu dem Gesamt-System gebildet, um darzustellen, inwieweit der „Rest" des Systems mit dem Fehlerpotenzial zu tun hat. Sollte der Fehler eher abgeschottet sein, wird dieser wohl kaum auffallen und Kosten in Bezug auf *Corrective Maintenance* erzeugen, ein stark frequentierter Datenbestand dagegen wird durchaus den Daten-Fehler irgendwann kenntlich machen. Um dies darzustellen, wird zuerst die universelle Funktion $N(d_i)$ genutzt, welche

das Nutzverhalten für einen Datenbestand berechnet:

$$N(d_x) = \sum_{y=1}^{m} (FP_{c_y} \cdot \overrightarrow{\varphi}_{c_y d_x} \cdot \overrightarrow{\varphi}')$$

Man kann erkennen, dass das Nutzungsverhalten eines Datenbestandes d_x aus den anliegenden Komponenten mit ihren *FP* sowie deren Umfang der Nutzung φ sowie dem Standardvektor φ' gebildet wird. Man erkennt diese Operation eventuell aus der *Adaptive Maintenance* wieder, nur dass da der Umfang der Nutzung pro Komponente gezählt wird, und nicht wie hier pro Datenbestand. Dies ergibt als Resultat von $N(d_x)$ eine Anzahl an *FP*, welche danach relativiert sind, inwieweit die einzelnen Nutzungen des Datenbestandes d_x ausfallen. Hierbei sieht man auch, dass das vorher angesprochene φ einspielt, weswegen dies nicht mehr in der eigentlichen Metrik auftaucht. Um nun das Verhältnis von Ziel-Datenbestand zum System zu bilden, wird das Verhältnis der Nutzung als Faktor in die Metrik eingebaut:

$$y_c = \sum_{i=1}^{m} (FP_{c_i} \cdot r(\lambda|c_i) \cdot \sum_{j=1}^{n} \sum_{k=1}^{n} (\sqrt{\overline{Rd}_{d_j d_k}} \cdot \frac{N(d_j) \cdot N(d_k)}{(\sum_{l=1}^{m} FP_{c_l})^2})) \text{ mit } \exists \varphi_{c_i d_j}$$

$$N(d_x) = \sum_{y=1}^{m} (FP_{c_y} \cdot \overrightarrow{\varphi}_{c_y d_x} \cdot \overrightarrow{\varphi}')$$

Das gebildete Verhältnis wird mit der bereits bestehenden Redundanz multipliziert. Die beiden Nutzungsfaktoren $N(d_j)$ und $N(d_k)$ werden multipliziert, und durch die gesamte Anzahl der im System enthaltenen *FP* (als Summe der *FP* aller Komponenten dargestellt) dividiert. Das Quadrat dieser Summe dient dazu, die Multiplikation der beiden Faktoren im Zähler des Bruches auszugleichen. Dabei wird die Eigenschaft genutzt, dass die Funktion $N(d_x)$ indirekt relativierte *FP* ausgibt, womit man den Bezug zu den Gesamt-*FP* herstellen kann. Betrachtet man die vermuteten Grenzwerte und beachtet, dass $N(d_x)$ maximal die Summe aller Gesamt-*FP* darstellen kann, wird man feststellen, dass bei voller Nutzung des Ziel-Datenbestandes das Verhältnis maximal 1 ergibt, und sobald eine Komponente nicht vollends den Datenbestand d_j oder d_k nutzt, ebenfalls das Verhältnis kleiner als 1 ergibt. Das Verhältnis des Fehlereinflusses kann 0 ergeben, wenn der Datenbestand d_j oder d_k niemals benutzt wird ($N(d_x) = 0$), womit auch kein Fehler durch *unmanaged* Datenredundanz $\overline{Rd}$ auftreten kann (im Sinne der *Corrective Maintenance* wohlgemerkt).

Fehlt nur noch ein Faktor, der wie in *Adaptive Maintenance* das Unternehmen darstellt, der sogenannte Unternehmensfaktor q_c. Dieser beinhaltet auch hier wieder die Faktoren des Unternehmens oder Projektes, welche nicht kausal erfasst werden können und dennoch größeren Einfluss haben. Das wären bei der *Corrective Maintenance* zum Beispiel die Qualität der Code-Dokumentation, die Verfügbarkeit von professionellen Entwicklern, dem Fachwissen der Mitarbeiter und andere Einflüsse. Es sei hierbei angemerkt, dass q_a nicht gleichbedeutend dem hier benutzen q_c sein muss. Auch q_c relativiert den resultierenden *ARI* in der Höhe, aber das bedeutet nicht, dass der zugehörige Faktor gleich der *Adaptive Maintenance* ist. Es ist späteren empirischen Ermittlungen vorbehalten, ein geeignetes q_c zu finden. Damit steht die finale Metrik

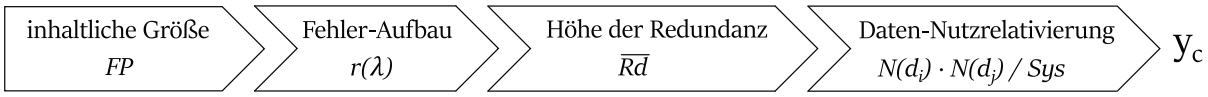


Abbildung 5.17: Die Zusammensetzung der *Corrective Maintenance*

mit folgender formaler Beschreibung:

$$y_c = \sum_{i=1}^{m}(FP_{c_i} \cdot r(\lambda|c_i) \cdot \sum_{j=1}^{n}\sum_{k=1}^{n}(\sqrt{\overline{Rd}_{d_j d_k}} \cdot \frac{N(d_j) \cdot N(d_k)}{(\sum_{l=1}^{m} FP_{c_l})^2})) \cdot q_c \text{ mit } \exists\varphi_{c_i d_j}$$

$$N(d_x) = \sum_{y=1}^{m}(FP_{c_y} \cdot \overrightarrow{\varphi}_{c_y d_x} \cdot \overrightarrow{\varphi}')$$

$$r(\lambda|c_i) = \frac{y - x}{y} \text{ mit } y = MaxIdx(\mathbb{E}) \text{ und } c_i \in \mathbb{E}_x^+; \text{ wenn } y = 0 \text{ dann } r(\lambda) = 0$$

Diese gesamte Metrik sollte geeignet das Verhalten der *Corrective Maintenance* auf Grundlage des Graphen wiedergeben können. Wie man erkennen kann, spielt die Anzahl der *Entities* keine Rolle, da aus der Praxis bekannt ist, dass ein Fehler in einem Code-Einzeiler genauso fatal sein kann, wie einige hundert Datensätze in einer Datenbank. Da der Umstand der verschieden großen Datenbestände bereits in *Rd* eingebaut ist, wird hierbei keine extra Berechnung durch *Entities* vorgenommen. Bei späteren, weiterführenden Arbeiten kann eine Vorgehensweise entwickelt werden, mit der man den Unterschied zwischen den *Entities* komplett negiert, der momentan noch durch die *Rd*-Metrik fest eingebaut ist.

Die Metrik ist im Grunde sehr variabel, was durch die Natur von unvorhersagbaren Fehlern begründet ist. Auch der Graph besitzt (derzeit) nur beschränkte Möglichkeiten, geeignete Messbarkeit für Fehler zu realisieren. Deswegen ist neben der Anpassung der Darstellung (des Graphen) auch die Metrik sehr variabel bezüglich $r(\lambda)$, $N(d_x)$ und der Relativierung von $\overline{Rd}$ (hier mit $\sqrt{\overline{Rd}}$) gestaltet.

In Abbildung 5.17 ist aufgezeigt, wie sich die größten Einflüsse der *Corrective Maintenance* zusammenstellen. Wieder ist die inhaltliche Größe der entsprechenden Komponenten relevant, was mit den Metriken des Fehlerverhaltens $r(\lambda)$ kombiniert wird. Zusätzlich wird die schädliche *unmanaged* Datenredundanz einbezogen, um dadurch den Wertverlust bestimmen zu können. Schlussendlich wird das Ergebnis noch einmal relativiert, je nachdem, wie stark die betroffenen Elemente im System integriert sind und damit Fehler, zum Beispiel durch Abhängigkeiten, verursachen könnten.

Preventive Maintenance Die letzte Metrik beschreibt die *Preventive Maintenance*, den Aufwand, welcher in proaktiven Austausch und Verbesserung von Komponenten gesteckt wird. Im Gegensatz zu der *Corrective Maintenance* wird hierbei kein Fehler gesucht und gefixt, sondern gezielt die Architektur und das Design verbessert und damit eventuell auftretende Fehler eliminiert, sowie bestehende *unmanaged* Datenredundanz beseitigt. Damit zielt diese Metrik darauf ab, einen roten Pfeil im Graphen (*unmanaged Redundancy*) in einen grünen Pfeil (synchronisierte Redundanz) abzuändern. Das ist nicht immer die beste Lösung für ein proaktives Vorgehen, da es durchaus sinnvoll wäre, den zusätzlichen Datenbestand einfach zu eliminieren und mit anderen zu vereinfachen. Das ist jedoch Aufgabe des „Reduce Complexity“-Prozesses, der in dieser Arbeit nicht thematisiert wird. Deswegen wurde entschieden, dass nur die einfachste Art

der *Preventive Maintenance* betrachtet wird (das oben beschriebene Umwandeln) und nicht weiter auf die Datenbestände und Komponenten eingegangen wird, weil dies zu schnell zu falschen Annahmen führen kann, die durch die Darstellung (das Graph-Modell) nicht gedeckt werden können.

Metrik-Idee

Die Intention bei dieser Metrik ist, einen Wert für den Aufwand zu erhalten, der dadurch entsteht, aus einer *unmanaged* eine *managed* Datenredundanz zu schaffen. Dies wird als Teil eines Rearchitectings und Refactorings des Systems betrachtet. Die Metrik soll Auskunft darüber geben, was es an Aufwand bedeutet, eine *unmanaged* Datenredundanz durch gezielte Synchronisation in einen nicht schadhaften Teil des Systems zu ändern. Dabei wird bewusst vernachlässigt, dass andere Lösungen (wie das Vereinigen oder Vereinfachen einiger Datenbestände) durchaus für einige Szenarien sinnvoller wären. Das wird auch mitunter dadurch begründet, dass eine Synchronisation so gut wie immer möglich ist, im Gegensatz zu spezielleren Lösungen. Der Unterschied zur *Corrective Maintenance* liegt hierbei darin, dass die schädliche *unmanaged* Datenredundanz ganz entfernt wird, was durchaus neue Methoden und Prozesse nach sich ziehen kann. Dagegen wird bei einer *Corrective Maintenance* nicht die Ursache entfernt, sondern nur der spezielle Teil einer Komponente, der einen Fehler verursacht.

Allerdings ist das Finden einer passenden Metrik für das Umwandeln eines Pfeiles im Graphen nicht einfach. Was spielt in einem proaktiven Austausch von *unmanaged Redundancy* in eine synchronisierte *managed* Datenredundanz eine Rolle? Der wichtigste Punkt ist vermutlich, dass diese Metrik nicht von Komponenten ausgeht, sondern von den Redundanzen selbst, womit sie sich von *Adaptive Maintenance* und *Corrective Maintenance* unterscheidet. Man bildet also eine Zähloperation pro *unmanaged Redundancy* und verbindet diese mit weiteren Faktoren. Einer davon ist eine Relativierung des Aufwandes bezüglich der Größe der redundanten Daten. Eine hart codierte Redundanz (also eine *Entity*) ist wesentlich einfacher zu handhaben als ein riesiges redundantes Datenbanksystem. Zwar wird der Strukturunterschied bereits von *Rd* erfasst, jedoch nur relativ und damit nicht ausreichend genug hinsichtlich der Größe des Datenbestandes. Ein weiterer Faktor wäre wieder die inhaltliche Größe der jeweiligen angeschlossenen Komponenten, welche für die aus der Metrik resultierende *eFP*-Größe des *ARI* benötigt werden. Letztendlich wäre auch hier eine Einbeziehung von Nutzungsverhalten sinnvoll, da der Aufwand, einen Datenbestand zu ersetzen, stark davon abhängt, inwieweit dieser auch benutzt wird. Ein schwach benutzter Datenbestand kann demzufolge leichter ausgetauscht werden als ein stark benutzter Datenbestand. Daher sollten auch hier wieder die vier generellen Eigenschaften zu *Nonnegativity*, *Null Value*, *Symmetry* und *Monotonicity* gelten.

Beginnt man mit dem ersten großen Schritt, der Zähloperation über alle an der *unmanaged Redundancy*'s beteiligten Komponenten, dann steht zuerst die folgende Metrik, welche zu Teilen aus allen zuvor behandelnden Metriken bekannt ist:

$$y_p = \sum_{j=1}^{n} \sum_{k=1}^{n} (N(d_j)^{\overline{Rd}_{d_j d_k}}) \text{ mit } \overline{Rd}_{d_j d_k} > 0$$

$$N(d_x) = \sum_{y=1}^{m} (FP_{c_y} \cdot \overrightarrow{\varphi}_{c_y d_x} \cdot \overrightarrow{\varphi}')$$

Man kann erkennen, dass wieder das Nutzungsverhalten des Datenbestandes $N(d_j)$ eine wichtige Rolle spielt, denn es ist immerhin genau der Datenbestand, welcher abgeändert werden muss, damit die *unmanaged Redundancy* entfernt wird. Die Funktion des Nutzungsverhaltens $N(d_j)$ wurde bereits in der Metrik für *Corrective Maintenance* eingeführt, weshalb hier auf eine genauere Analyse verzichtet wird. Wichtig ist zu verstehen, dass das Nutzungsverhalten eines Datenbestandes widerspiegelt, inwieweit ein Datenbestand tatsächlich mit den Operationen von *CRUD* benutzt wird. Eine häufig frequentierte Datenbank hat demnach einen höheren Nutzungsfaktor als ein Backup, was üblicherweise keinerlei Benutzung aufweisen kann, da dies nicht dem Sinn eines Backups entspricht. Kennt man das Nutzungsverhalten und damit die Abhängigkeiten eines Datenbestandes, kann man (wie in der *Corrective Maintenance*) abschätzen, inwieweit es schwierig wird, die *unmanaged Redundancy* zu entfernen.

Zudem kann man erkennen, dass die Redundanz $\overline{Rd}$ exponentiell in die Metrik einfließt, da man davon ausgehen kann, dass die Komplexität im System ansteigt, je höher die Redundanz ist. Im Gegensatz zu der *Corrective Maintenance*, welche eine Wurzelfunktion beinhaltet, wird bei der *Preventive Maintenance* davon ausgegangen, dass nicht ein Fehler behoben wird (welcher eine obere Grenze an Ursachen ausweist), sondern dass Abhängigkeiten und Komplexität die Lage erschweren, die Architektur und das Design im Sinne der *Preventive Maintenance* zu verbessern. Gleichzeitig spiegelt es die erhöhte Komplexität bei hoher Redundanz wieder, da dadurch mehr *Frames* synchronisiert werden müssen, welche durchaus je andere Prozesse der Synchronisation erfordern. Der Grund, warum eine exponentielle Funktion gewählt wurde, liegt an der exponentiellen Natur von Phänomenen, welche Komplexität beinhalten [6]. Ein Beispiel für solche exponentiellen Erscheinungen ist das Scheduling von Ressourcen in einem Betriebssystem. Es wird exponentiell aufwendiger, die Ressourcen sinnvoll zu verteilen, je mehr Anfragen nach Ressourcen anstehen. Ein weiteres Beispiel ist die Staubildung, in der die Wartezeit (der „Aufwand“) exponentiell ansteigt (beziehungsweise auch die Geschwindigkeit der Autos exponentiell abnimmt), je mehr Autos auf einer Strecke fahren. An diesen Beispielen mit der exponentiellen Erscheinung wurde die hier entstandene Metrik angelehnt, mit der Theorie, dass der Aufwand der *Preventive Maintenance* exponentiell ansteigt, je mehr *unmanaged Redundancy* vorhanden ist. Man könnte nun bereits einen weiteren Schritt hin zur finalen Metrik gehen, jedoch gibt es bereits jetzt schon ein erstes Problem mit der Metrik. Durch mathematische Definition einer Exponentialfunktion ist bei einem Null-Exponenten das Resultat $x^0 = 1$ definiert. Das bedeutet in diesem Fall für den Aufwand y_p, dass, wenn keinerlei *unmanaged Redundancy* $\overline{Rd}$ anliegt, dennoch $N(d_j)^0 = 1$ ergibt, was falsch wäre, da dies bereits einen positiven *ARI* darstellen würde. Um das zu umgehen, wird die Bedingung $\overline{Rd}_{d_j d_k} > 0$ hinzugefügt. Damit ist die Metrik für $\overline{Rd} = 0$ auf 0 definiert. Im Graphen ist das Problem nicht ganz so schwierig, da dort sowieso nur eine *Rd*-Kante existiert, wenn deren entsprechender Wert größer 0 ist. Dagegen ist es kein Problem, wenn ein Datenbestand nicht genutzt wird und damit $N(d_j) = 0$ ist, da immer $0^{\overline{Rd}} = 0$ ergibt, unabhängig des Wertes für $\overline{Rd}$.

Damit sind bereits drei wichtige Faktoren einbezogen, zum einen die inhaltliche Größe von Komponenten und deren Nutzungsverhalten über die Nebenfunktion $N(d_j)$ sowie die Redundanz *Rd*. Ein wichtiger Faktor fehlt derzeit noch, die Relativierung der strukturellen Größe. Ein Datenbestand wird theoretisch gesehen weniger Aufwand verursachen, als ein Datenbestand, der als Ankerpunkt im System dient. Um dies zu verdeutlichen, wird ein zusätzlicher Faktor eingeführt, der allgemein als *Median* [63] bekannt ist:

$$\tilde{x} = \begin{cases} x_{\frac{n+1}{2}}, & \text{wenn } n \text{ ungerade,} \\ \frac{1}{2}(x_{\frac{n}{2}} + x_{\frac{n}{2}+1}), & \text{wenn } n \text{ gerade.} \end{cases}$$

Der Median bestimmt aus einer geordneten Liste genau den Mittelpunkt, beziehungsweise dessen Wert. Damit ist sichergestellt, dass eventuelle Ausreißer am „Rand“ der Liste nicht den allgemeinen Wert verändern, wie es beim Durchschnitt der Fall ist. Dazu werden alle verfügbaren Werte ihrer Größe nach sortiert, mit Indizes versehen (in der Formel als Fußnote) und dann aus der Reihe das mittlere Element gewählt. Für den Fall, dass es genau ein mittleres Element gibt (wenn n als maximaler Index ungerade ist), wird das Element $x_{\frac{x+1}{2}}$ gewählt. In einer beispielhaften Reihe von $[1, 3, 4, 4, 18]$ wäre dann $\tilde{x} = 4$, ohne dass die 18 den Wert „verfälscht“. Sollte einmal n gerade sein, dann werden die beiden Werte gewählt, welche den Mittelpunkt umschließen und davon der Mittelwert gebildet. In der beispielhaften Reihe von $[1, 2, 5, 8]$ wäre dadurch $\tilde{x} = \frac{1}{2}(2 + 5) = 3.5$. Der Median ist immer dann geeignet, wenn zu erwarten ist, dass einzelne, zu große Abweichungen einen realistischen Wert verfälschen. In dieser Arbeit ist $\tilde{x}$ ein bestimmter E-Wert in einer geordneten Reihe an allen im System verfügbaren E-Werten (also aller Datenbestandsgrößen in *Entities*). Das führt dazu, dass neue, extrem abweichende Datenbestandsgrößen vorerst unbeachtet bleiben, sodass sich sehr gut ein Mittelmaß des Systems einstellen kann, an dem sich der oben beschriebene Aufwand messen lassen kann. Ist der betrachtete Datenbestand besonders klein im Vergleich zu allen anderen (also im Vergleich zu dem Median), dann kann man davon ausgehen, dass relativ betrachtet weniger Aufwand betrieben werden muss. Ist er besonders groß, verhält es sich analog. Dieses Verhältnis doppelt sich etwas mit der strukturellen „Gleichheit“ in der *Rd*-Metrik, jedoch ist es sinnvoll, dieses neue Verhältnis trotzdem zu nutzen, da dadurch ein Gefühl dafür entwickelt wird, wie viel Aufwand tatsächlich entsteht, unabhängig der wirklich kopierten Daten. Dies wird dadurch gestützt, dass der Unterschied in der *Rd*-Metrik eher absolut gezählt wird (also durch eine tatsächliche Anzahl an *Entities*), dieser Median-Faktor jedoch relative Unterschiede betrachtet. Baut man dieses Verhältnis in die Metrik ein, dann erhält man folgendes:

$$y_p = \sum_{j=1}^{n}\sum_{k=1}^{n}\left(\frac{E_{d_j}}{\tilde{E}} \cdot N(d_j)^{\overline{Rd}_{d_j d_k}}\right) \text{ mit } \overline{Rd}_{d_j d_k} > 0$$

$$N(d_x) = \sum_{y=1}^{m}(FP_{c_y} \cdot \overrightarrow{\varphi}_{c_y d_x} \cdot \overrightarrow{\varphi}')$$

$$\tilde{E} = \begin{cases} E_{\frac{n+1}{2}}, & \text{wenn } n \text{ ungerade,} \\ \frac{1}{2}(E_{\frac{n}{2}} + E_{\frac{n}{2}+1}), & \text{wenn } n \text{ gerade.} \end{cases}$$

Damit sind theoretisch alle großen Einflussfaktoren enthalten, fehlt nur noch der obligatorische Unternehmensfaktor q_p, welcher die nicht erfassbaren Faktoren erfasst und damit das Resultat der Metrik in der Höhe relativiert. Wieder sei hier zu beachten, dass q_p nichts mit den anderen beiden Werten q_a und q_c zu tun hat, sondern ein eigenständiger Wert ist, der zuvor empirisch ermittelt werden muss. Zudem sollte die Bedingung gelten, dass $\overline{Rd}_{d_j d_x} > 0$ ist, damit wirklich nur *unmanaged Redundancy*'s erfasst werden. Damit ist die finale Metrik wie folgt definiert:

$$y_p = \sum_{j=1}^{n}\sum_{k=1}^{n}\left(\frac{E_{d_j}}{\tilde{E}} \cdot N(d_j)^{\overline{Rd}_{d_j d_k}}\right) \cdot q_p \text{ mit } \overline{Rd}_{d_j d_k} > 0$$

$$N(d_x) = \sum_{y=1}^{m}(FP_{c_y} \cdot \overrightarrow{\varphi}_{c_y d_x} \cdot \overrightarrow{\varphi}')$$

$$\tilde{E} = \begin{cases} E_{\frac{n+1}{2}}, & \text{wenn } n \text{ ungerade,} \\ \frac{1}{2}(E_{\frac{n}{2}} + E_{\frac{n}{2}+1}), & \text{wenn } n \text{ gerade.} \end{cases}$$

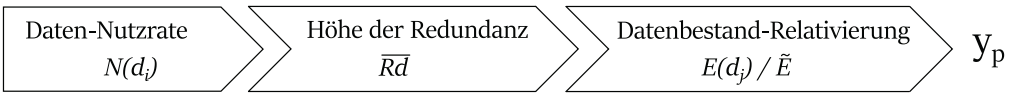


Abbildung 5.18: Die Zusammensetzung der *Preventive Maintenance*

Diese Metrik für *Preventive Maintenance* ist damit theoretisch geeignet, das Beseitigen einer *unmanaged Redundancy* einzuschätzen. Hierbei liegen die variablen Teile der Metrik auf der Funktion $N(d_j)$, dem Unternehmensfaktor q_p sowie einer Erweiterung des Verhältnisses der Größe. Zudem können weitere Faktoren benutzt werden, um potenzielle Fehleinschätzungen zu unterbinden. Ein Problem könnte dabei auftreten, dass das Größen-Verhältnis der Datenbestände nicht ausreichend relativiert ist. Ein weiterer Kritikpunkt könnte der Fakt sein, dass eine exakte Kopie ($\overline{Rd} = 1$) eines Datenbestandes sehr hohen *ARI* verursacht, wobei man in der Praxis einfach einen der beiden Datenbestände eliminiert und nur den anderen nutzt. Dabei ist zu beachten, dass es bei der hier definierten *Preventive Maintenance* nicht um eine „Vereinfachung“ geht, sondern nur um das Umwandeln einer *unmanaged Redundancy* in eine synchronisierte Redundanz, was ein komplett anderes Thema darstellt. Dass man am Ende mit „Vereinfachung“ durch Zusammenfassen („Reduce Complexity“-Prozess) besser bezüglich der *ARI* stehen könnte, wird hier nicht bestritten, jedoch ist im Sinne des „Synchronizität herstellen“ der extrem hohe *ARI* bei hoher Redundanz (bis hin zur exakten Kopie) durchaus berechtigt. Eine Vereinfachung wäre das Vorgehen eines anderen Architektur-Prinzips.

In Abbildung 5.18 sind, wie bei den anderen beiden Wert-Metriken, noch einmal alle großen Einflussfaktoren aufgezeigt. Insgesamt sind dies weniger als bei den anderen beiden Metriken, da die *Preventive Maintenance* „nur“ dazu dient, die *unmanaged* Datenredundanz in eine *managed* Datenredundanz umzuwandeln, nicht das komplette System zu überarbeiten (was durchaus sehr viele Einflüsse hätte). Deshalb wird hier von dem jeweiligen Nutzungsverhalten eines Datenbestandes $N(d_i)$ ausgegangen, der das Ziel der kopierten Daten darstellt. Dem wird die Höhe der Redundanz hinzugerechnet, um den Mehraufwand der Redundanz-Umwandlung abzuschätzen. Zum Schluss wird der entsprechende Datenbestand relativiert, um festzustellen, ob es sich um den Haupt-Datenbestand im System oder um ein kleines Code-Schnipsel handelt, was Auswirkungen auf die Einfachheit der Beseitigung der *unmanaged* Datenredundanz besitzt.

Zusammenfassung Die drei Wert-Metriken für *Adaptive*, *Corrective* und *Preventive Maintenance* sind nun festgelegt und dienen der Messbarkeit für den Systemwert. Die Funktionalität der einzelnen Formeln unterscheiden sich, jedoch ist ihnen allen gleich, dass als Resultat jeweils ein *ARI* gegeben ist, einer für jeden der drei Bereiche. In den Metriken wurden die bekannten und großen Einflussfaktoren einbezogen, welche in dem jeweiligen Bereich eine größere Rolle spielen und es wurde versucht, das praktische Verhalten in den Metriken widerzuspiegeln. Dabei stehen die drei Metriken, entgegen der in Kapitel 2.1.3 „Der Wert eines Systems“ genannten Zusammenhänge, jeweils unabhängig dar. Keine der Metrik beeinflusst die andere in irgendeiner Form. Das wurde deshalb so umgesetzt, da eine gegenseitige Beeinflussung zu einer stark erhöhten Komplexität der Metriken selbst geführt hätte und eine sinnvolle theoretische Grundlage damit nicht möglich gewesen wäre. Jedoch ist es denkbar, dass zukünftige Arbeiten weitere Einflussfaktoren einführen, welche auch die gegenseitige Beeinflussung der drei Metriken beinhalten könnten. Dafür wäre es notwendig, neue Probleme zu lösen, wie zum Beispiel die Reihenfolge der Abarbeitung oder der Handhabung einer ungewollten gegenseitigen „Überlagerung“ oder „Stapelung“ der Werte. Das, was die drei Metriken in dieser Arbeit vereint, sind die unterschiedlichen Bereiche, welche sie abdecken. Durch eine einfache Addition der Resultate y_a, y_c und y_p

ergibt sich somit ein Gesamtbild des Systems. Konkret bedeutet dies, dass drei Werte für den *ARI* generiert werden, welche zusammengefasst den Mehraufwand für das System bestimmen.

Im nächsten Kapitel geht es um das System an sich, wie die drei Metriken zusammenspielen und wie durch die gewonnene Aussage über den *ARI* und die *eFP* der Wertverlust des Systems aufgezeigt werden kann. Es werden die einzelnen Parameter betrachtet, die „Wert-Formel“ wird aufgestellt und erläutert, das dynamische Verhalten bei veränderlichen Parametern analysiert sowie eine Fehleranalyse gegeben, um die Probleme aufzuzeigen, welche sich mit den erstellten Graph- und Wert-Metriken ergeben, denn es wird in dieser Arbeit nicht der Anspruch erhoben, die perfekte Lösung gefunden zu haben.

5.4 III: Der Wert des Systems

Dieses Kapitel zum dritten Block soll nun den Abschluss der kausalen Kette von Zusammenhängen darstellen, wie eine Datenredundanz sich auf dem Wert der Software auswirkt. Im ersten Block wurden die Eigenschaften des Systems aufgezeigt, indem eine Darstellung geschaffen wurde, welche weitere Schlüsse zulässt. Im zweiten Block wurde mithilfe der Messbarkeit der Zusammenhang geschaffen, inwieweit die Eigenschaften der Darstellung, und damit des Systems, ausgewertet werden können. Jetzt folgt dazu im dritten Block die Gesamtansicht der Teile, die globale Zusammenfassung, mit dem Resultat, insgesamt einen kausalen und quantifizierten Weg der Zusammenhänge gefunden zu haben. Dass solch ein Zusammenhang von Struktur und Wert existiert, wurde in den ersten Kapiteln dieser Arbeit schon angedeutet und ist in Fachkreisen bekannt. Bislang fehlten Kenntnisse, wie genau dieser Zusammenhang verläuft.

Konkret sollen hier in diesem Kapitel das Zusammenspiel der einzelnen Wert-Metriken betrachtet und eine sogenannte „Wert-Formel“ entwickelt werden. Zuvor müssen jedoch einige Aspekte betrachtet werden, zum Beispiel wie genau sich die Parameter im zeitlichen Verlauf verhalten, wie die Metriken vereinbar sind oder welche Fälle ausgeschlossen werden müssen. Besonders wichtig ist hierbei die Frage nach dem zeitlichen Verhalten in Bezug auf das Erfassen der Metriken. Wie genau sollte man *Adaptive Maintenance* an Komponenten messen, erneut bei jeder neuen Erweiterung oder steht dessen Wert einmalig fest? Wenn solche grundlegenden Fragen geklärt sind, wird das Ende des Nachweises mit den folgenden Schritten erreicht: zuerst wird dargestellt, inwiefern das Resultat der Wert-Metriken als *ARI* interpretiert werden kann, danach wird der Verfall des Systems untersucht, als inverses Element zu Werterhalt und -Steigerung. Die entsprechende Vorgehensweise wird sein, die „Wert-Formel“ aufzustellen, dies in die nötigen Eigenschaften *TtM* und *DevC* umzuwandeln und damit den Wertverlust des Systems darzustellen, was das Ende des Nachweises bildet. In Abbildung 5.19 ist ein Ausschnitt aus der bereits bekannten Abbildung 5.9 dargestellt. Diese beiden letzten Schritte sollen mit dem nun folgenden Block abgedeckt werden.

Um die Übersichtlichkeit zu wahren, sind noch einmal alle drei Metriken in Abbildung 5.20 aufgelistet und nummeriert, um darauf referenzieren zu können. Der Aufbau der drei Metriken ist folgend nochmals kurz dargestellt.

5.1 Die *Adaptive Maintenance* besteht aus dem Zählen aller *FP*'s in Form von $C(c_i)$, das heißt jeder Komponente. Von jeder Komponente aus wird deren Nutzungsbeziehung φ zu einem Datenbestand analysiert und mit dessen Redundanz $\overline{Rd}$ multipliziert. Eine Ausnahme $\delta_{c_i d_j}$ wird genau dann eingeräumt, wenn der Datenbestand in der gleichen Erweiterung wie die Komponente entsteht, als kurzzeitiger Vorteil durch einen Shortcut.

5.4 Die *Corrective Maintenance* setzt ebenfalls auf die *FP* der Komponenten. Es wird die

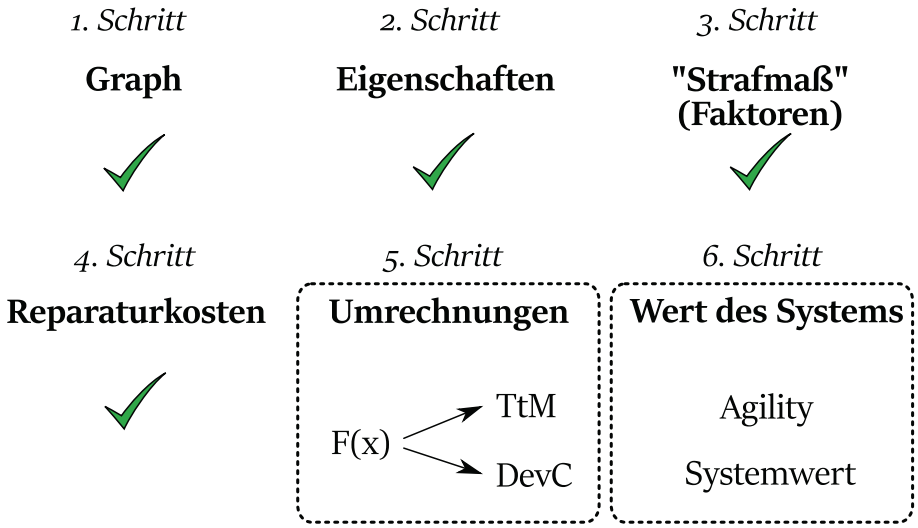


Abbildung 5.19: Die letzten Schritte der kausalen Kette im Nachweis

Funktion $r(\lambda)$ benutzt, um das zeitliche Verhalten zu simulieren. Anschließend werden die „angeschlossenen“ Redundanzen $\overline{Rd}$ per Wurzelfunktion hinzu berechnet. Als Abschluss wird noch das Verhältnis des betreffenden Nutzungsverhaltens $N(d_i)$ und der Gesamtheit an FP $\sum FP$ zu jeder Redundanz multipliziert, welches darstellt, wie relevant der Datenbestand im Vergleich zu allen anderen ist, und wie wahrscheinlich eine Nutzung des Datenbestandes einen Fehler auslösen kann.

5.7 Die *Preventive Maintenance* berechnet sich durch die einzelne Analyse aller Datenbestände, da hier die Kosten für die Umwandlung von *unmanaged Redundancy*'s in synchronisierte Redundanz eingeschätzt werden. Der erste Faktor ist ein Verhältnis, welches beschreibt, inwieweit die Größe des Ausgangsdatenbestandes eine Rolle spielt (im Vergleich zum Median). Der Hauptfaktor besteht aus dem Nutzungsverhalten $N(d_j)$ des betrachteten Datenbestandes, welches dann mit der Redundanz $\overline{Rd}$ in exponentieller Funktion steht.

5.4.1 Ermittlungen der Parameter

Wie bei der Erstellung der Parameter erläutert, gibt es bei fast allen Aspekten der Metriken eine gewisse Variabilität hinsichtlich bestimmte Faktoren. Dafür gibt es relativ wenige Parameter, welche unbestimmt bleiben und besetzt werden müssen. In diesem Abschnitt soll noch einmal geklärt werden, welcher Teil der Metriken mit welchen Werten angenommen wird, was dessen Grenzen sind und welchen Einfluss diese haben. Es sei hier betont, dass nicht auf die variablen Parts eingegangen wird, wie zum Beispiel das Austauschen eines Faktors gegen einen besser geeigneten, sondern lediglich die offenen Parameter betrachtet werden.

Bei der *Adaptive Maintenance* sind vier Parameter verbaut, wie man in der Metrik 5.1 erkennen kann. Die ersten beiden Parameter sind FP_{c_i} und $\overline{Rd}_{d_j d_k}$, welche auch in jeder weiteren Metrik vorkommen werden, wobei sie bei der *Adaptive Maintenance* die FP_{c_i} als $C(c_i)$ eingehen. Diese beiden Parameter sind einfach aus dem Graphen ersichtlich und gehören zu dem Grundgerüst des Nachweises, da alles auf den FP als Aufwand und Rd als Ursache des Schadens aufbaut. Die FP sind direkt in den Komponenten als Attribut enthalten, die Rd dagegen als Annotation an den Pfeilen des Graphen ablesbar. Der dritte Parameter ist φ', welcher laut

Adaptive Maintenance

$$y_a = \sum_{i=1}^{m}(C(c_i) \cdot \sum_{j=1}^{n}(\delta_{c_i d_j} \cdot \overrightarrow{\varphi}_{c_i d_j} \cdot \overrightarrow{\varphi}' \cdot \sum_{k=1}^{n} \overline{Rd}_{d_j d_k})) \cdot q_a \tag{5.1}$$

$$\delta_{c_i d_j} = \begin{cases} 1, & \text{wenn } c_i \in \mathbb{E}_x^+ \text{ und } d_j \notin \mathbb{E}_x^+ \text{ mit } x \in \mathbb{N}, \\ 0, & \text{sonst.} \end{cases} \tag{5.2}$$

$$C(c_i) = \frac{FP_{c_i}}{\sum \mathbb{E}_y^+} \text{ mit } c_i \in \mathbb{E}_y^+ \tag{5.3}$$

Corrective Maintenance

$$y_c = \sum_{i=1}^{m}(FP_{c_i} \cdot r(\lambda|c_i) \cdot \sum_{j=1}^{n}\sum_{k=1}^{n}(\sqrt{\overline{Rd}_{d_j d_k}} \cdot \frac{N(d_j) \cdot N(d_k)}{(\sum_{l=1}^{m} FP_{c_l})^2})) \cdot q_c \text{ mit } \exists \varphi_{c_i d_j} \tag{5.4}$$

$$N(d_x) = \sum_{y=1}^{m}(FP_{c_y} \cdot \overrightarrow{\varphi}_{c_y d_x} \cdot \overrightarrow{\varphi}') \tag{5.5}$$

$$r(\lambda|c_i) = \frac{y - x}{y} \text{ mit } y = MaxIdx(\mathbb{E}) \text{ und } c_i \in \mathbb{E}_x^+; \text{ wenn } y = 0 \text{ dann } r(\lambda) = 0 \tag{5.6}$$

Preventive Maintenance

$$y_p = \sum_{j=1}^{n}\sum_{k=1}^{n}(\frac{E_{d_j}}{\tilde{E}} \cdot N(d_j)^{\overline{Rd}_{d_j d_k}}) \cdot q_p \text{ mit } \overline{Rd}_{d_j d_k} > 0 \tag{5.7}$$

$$N(d_x) = \sum_{y=1}^{m}(FP_{c_y} \cdot \overrightarrow{\varphi}_{c_y d_x} \cdot \overrightarrow{\varphi}') \tag{5.8}$$

$$\tilde{E} = \begin{cases} E_{\frac{n+1}{2}}, & \text{wenn } n \text{ ungerade,} \\ \frac{1}{2}(E_{\frac{n}{2}} + E_{\frac{n}{2}+1}), & \text{wenn } n \text{ gerade.} \end{cases} \tag{5.9}$$

Abbildung 5.20: Die Wert-Metriken

Metrik-Erstellung initial mit

$$\varphi' = \begin{pmatrix} 0.25 \\ 0.10 \\ 0.40 \\ 0.25 \end{pmatrix}$$

angegeben wurde. Das φ' stellt sicher, dass verschiedene *CRUD*-Operationen unterschiedlich gewichtet werden, je nachdem, wie viel Schadenspotenzial diese besitzen. Die Werte von φ', welche hierbei vorliegen, sind theoretische ermittelt und werden als Anlaufpunkt für die Gewichtung dienen. Man kann durchaus infrage stellen, ob ein *Create* und ein *Delete* mit 0.25 gleichzusetzen ist oder ob ein *Update* viermal mehr schädlicher ist als ein *Read*. Doch das ist nicht relevant, denn dieser Parameter φ' geht einmalig ein und erzeugt keine weiteren Abhängigkeiten. Sollten zukünftig bessere empirische Werte ermittelt werden, können die Werte von φ' dahingehend leicht angepasst werden. In dieser Arbeit vorerst mit dieser Belegung der Werte φ'_C, φ'_R, φ'_U und φ'_D weitergearbeitet. Der letzte Parameter ist der Unternehmensfaktor, der das Resultat der Metrik in der Höhe relativiert. Sollte sich ein zu geringer oder zu hoher Wert für *ARI* ergeben und sich damit als unrealistisch herausstellen, kann dementsprechend der Faktor q_a angepasst werden. Für diesen Parameter gibt es keinen allgemeingültigen Wert, selbst nach empirischen Messungen muss dieser immer dann angepasst werden, wenn sich der Kontext des Systems ändert. Ein anderes Unternehmen, ein anderes Projekt, eine andere Technik oder auch andere Entwickler erfordern ein Anpassen des Unternehmensfaktors, welcher (entgegen seiner Benennung) auch durch Einflüsse außerhalb des Unternehmens veränderbar ist. Nun kann es den Anschein haben, dass dieser Faktor q_a so stark ist, dass die Metrik dahinter ihren Sinn verliert, was jedoch nicht wirklich der Fall ist. Die durch die Metrik gegebene Aufstellung der Faktoren bleibt dahingehend gleich, dass die Auswirkungen der einzelnen Faktoren der Metrik sowie der Verlauf deren Werte in relativer Beziehung das Verhältnis wahren. Durch eine Abänderung von q_a wird es nicht möglich sein, den Einfluss von *Rd* auf die allgemeine Metrik zu verändern, jedoch zu starke Gesamtauswirkung aller Faktoren zu relativieren. Das führt dazu, dass die Kausalität in dem Nachweis erhalten bleibt und maximal die Quantifizierung ungenau wird, wobei das Verhalten der Quantifizierung ebenfalls unverändert bleiben sollte. In dieser Arbeit wird $q_a = 1$ angenommen, da weder empirische Studien vorliegen, noch andere Werte für q_a sinnvoller erscheinen. Dadurch erkennt man ebenfalls die konkreten Auswirkungen ohne Relativierung unbekannter Phänomene durch den Unternehmensfaktor.

Für die zweite Metrik der *Corrective Maintenance* sind größtenteils die gleichen Parameter FP, Rd, φ' und q_c enthalten, weshalb hier nicht noch einmal darauf eingegangen wird, da sie sich nicht von den Parametern der *Adaptive Maintenance* unterscheiden. Zu beachten ist, dass q_c einen komplett anderen Wert aufweisen kann als q_a, jedoch in dieser Arbeit auch hierbei $q_c = 1$ angenommen wird. Ein neuer Parameter ist, wie in Metrik 5.4 zu erkennen, der Funktionsparameter $r(\lambda)$, welcher in Metrik 5.6 dargestellt ist. Dieser wurde bereits im vorherigen Kapitel erläutert, seine Bedeutung als Parameter ist dabei jedoch sehr groß. Der Funktionswert von $r(\lambda)$ wird direkt in den Wert von FP eingerechnet, was nicht nur eine Höhenrelativierung wie q als Faktor ergibt, sondern einen zeitlichen Verlauf in die Metrik einbringt. Mit einer geeigneten Funktion $r(\lambda)$ kann die FP stark variabel gestaltet werden. Aber auch das unterliegt letzten Endes empirischen Ermittlungen, weswegen in dieser Arbeit die naive Funktion 5.6 gewählt wurde, welche die theoretische Situation widerspiegelt, dass ein Fehler über Zeit mehr und mehr ersichtlich wird und seine Auswirkungen hat. Damit besitzt $r(\lambda)$ in dieser Arbeit die untere Grenze von 0 und eine obere Grenze von 1, was bedeutet, dass die zugehörigen FP von gar nicht

bis komplett eingehen. $r(\lambda)$ kann jedoch auch so gewählt werden, dass im Laufe der Zeit die *FP* abgeschwächt werden, weil dem Entwickler der Fehler bewusst ist und er damit umgehen kann (anstatt ihn zu beseitigen). Aber auch ein Mittelweg wäre denkbar.

In der dritten Metrik für *Preventive Maintenance*, zu sehen in Metrik 5.7, sind außer den Standard-Parametern FP, Rd, φ' und q_p keine weiteren Parameter enthalten. Hier ist wieder der Unterschied von q_p zu q_a und q_c zu beachten, wobei in dieser Arbeit q_p wieder normal mit $q_p = 1$ belegt ist.

5.4.2 Die Wert-Formel

Betrachtet man die drei Metriken bezüglich ihrer Vereinbarkeit, kann man folgende Erkenntnisse feststellen:

1. viele Teile der Metriken gleichen sich auf die eine oder andere Art und Weise,
2. es gibt nur zwei Ausgangspunkte der Zähloperation: Komponenten und Datenbestände,
3. einige Werte könnten einmal berechnet werden und als weitere Annotation von Elementen im Graphen auftauchen, und
4. die Metrik für *Corrective Maintenance* ist die Einzige, welche von der Zeit abhängig ist.

Diese Eigenschaften kann man sich für eine einfachere Berechnung aller drei Metriken zunutze machen, wobei sich im Rahmen dieser Arbeit dagegen entschieden wurde, alle drei Metriken zu vereinen und stattdessen getrennt voneinander jeweils auf ein System anzuwenden, um die unterschiedlichen Intentionen der Metriken zu unterstreichen und auch partielle Sichtweisen (zum Beispiel nur eine Metrik zu nutzen oder eine einzelne Metrik abzuändern) zu unterstützen. Elemente der Metriken, welche wiederholt benutzt werden, sind Faktoren wie das Nutzungsverhalten $N(d_i)$, der Umfang der Nutzung multipliziert mit der Standard-Gewichtung $\varphi_{c_i d_j} \cdot \varphi'$ oder auch die Summe aller *Entities* im System mit $\sum_{l=1}^{n} E_l$. Diese und andere Teile kann man, einmal berechnet, für gewisse Elemente im Graphen wiederverwenden. Es folgen nun die drei Schritte, die den Nachweis komplettieren: der System-*ARI*, die Systemwert-Ermittlung k und schlussendlich die Abbildung auf die beiden Einheiten *TtM* und *DevC*, beziehungsweise auf die *Agility* des Systems.

Der ARI des Gesamtsystems Berechnet man für alle drei Metriken den jeweiligen *ARI* y_a, y_c und y_p für einen Systemzustand aus, dann kann man den summierten *ARI* für das Gesamtsystem im fixen Zustand $\mathbb{E}_x^+$ berechnen durch

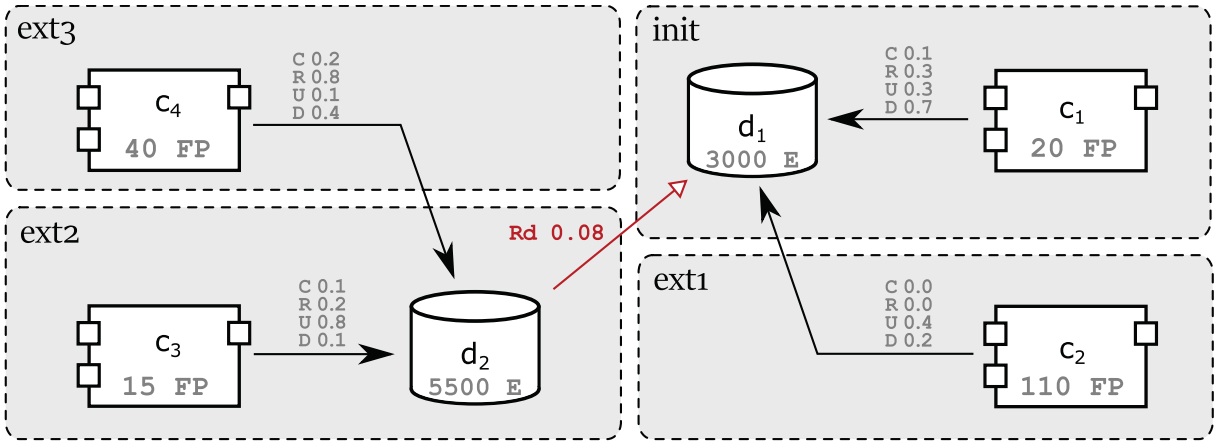
$$y(\mathbb{E}) = y_a(\mathbb{E}) + y_c(\mathbb{E}) + y_p(\mathbb{E})$$

Man kann erkennen, dass dieser Gesamtaufwand an *ARI* (dargestellt als $y(\mathbb{E})$) einmalig pro Zustand berechnet werden kann. Das mag auf dem ersten Blick bei der *Adaptive* und *Preventive Maintenance* einen Sinn ergeben, da beide einmalige Aktionen darstellen: das Entwickeln neuer und das Reparieren alter Elemente. Die Problematik ergibt sich jedoch durch die *Corrective Maintenance*, bei der man im Allgemeinen die Vorstellung besitzt, dass ein Fehler solange sich erhöhenden „Schaden“ in Form von *ARI* verursacht, bis er behoben ist. Da in der Darstellung dieses Nachweises keine echte Zeitachse vorhanden ist (außer die diskrete Erweiterung des Systems), kann dieses Verhalten nicht modelliert werden. Dennoch stellt sich die Frage, ob dieses Verhalten auch simuliert werden muss, da auch die *Corrective Maintenance* als fixer

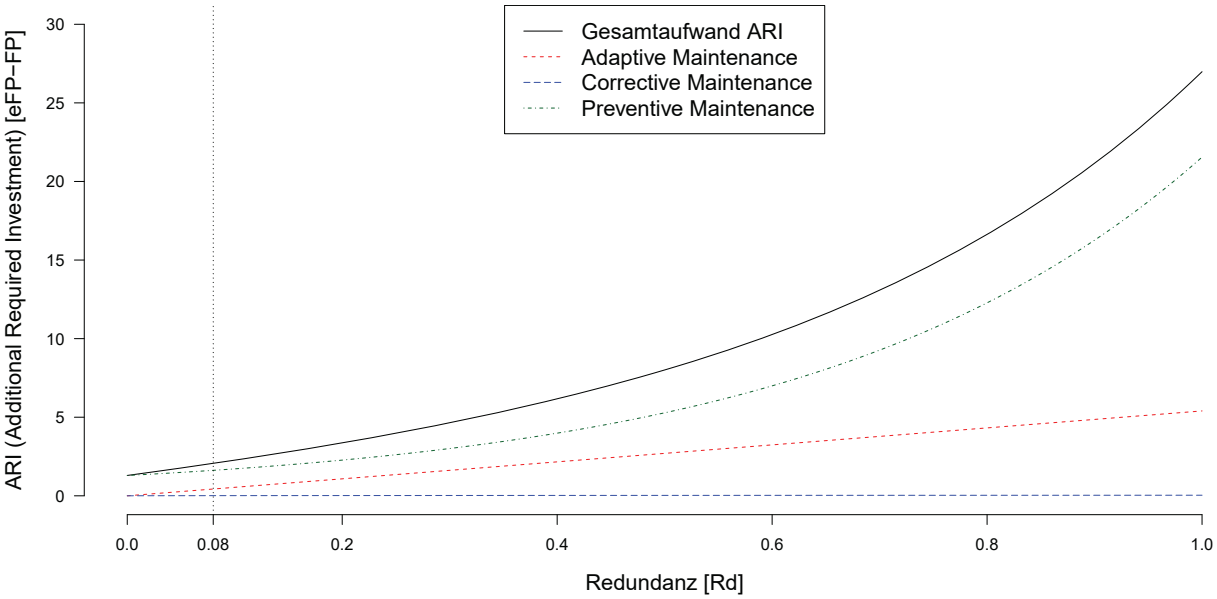
Faktor angesehen werden kann. Dies ist hierbei die einzige Möglichkeit, entsprechende Kosten für Fehlerkorrekturen mit diesem Modell zu vereinen. Durch die darin enthaltene Funktion $r(\lambda)$ ist eine Steigerung durch Nichtbeachtung bereits enthalten und bewirkt, dass sich der totale *ARI* durch *Corrective Maintenance* im Laufe der Zeit mit jeder Systemerweiterung verändert (in diesem Falle erhöht), womit ein Fehler-ähnliches Verhalten angedeutet wird, solange sich die *unmanaged Redundancy* im System befindet. Es sei hier noch einmal darauf hingewiesen, dass nur die funktionalen Fehlerauswirkungen (sowie deren Kosten für Behebung) betrachtet werden, nicht jedoch die Auswirkungen des Fehlers auf Reputation, Schadenszahlungen, Ersatz, rechtliche Folgen und anderes, was die Kosten eines Fehlers in immense Höhen treiben kann.

Durch den Gesamt-*ARI* ergibt sich der Aufwand, welcher bewältigt werden muss, um das System zu benutzen. Dabei wird keinerlei neue Funktionalität umgesetzt, sondern diese *eFP* als reine Kostensache angesehen. Durch die drei einzelnen Metriken erhält man eine Abdeckung der Wert-spezifischen Bereiche, welche in Kapitel 2.1.3 „Der Wert eines Systems“ eingeführt wurden. Durch negative Architekturentscheidungen in Form von *unmanaged Redundancy* wurde *Technical Debt* verursacht, welche Folge- und Bereinigungskosten nach sich zieht. Diese Kosten liegen nun in *ARI* für das Gesamtsystem vor.

In Abbildung 5.21 wurde für ein Beispielsystem die Berechnung der drei Metriken vorgenommen. Man kann in diesem Beispiel erkennen, dass die drei Metriken verschiedene Verläufe in der Höhe abhängig von der Redundanz $\overline{Rd}$ aufweisen, was in den Kurven in Abbildung 5.21b dargestellt ist. Für dieses Beispiel kam das in 5.21a abgebildete System als Berechnungsgrundlage zum Einsatz. Dieses System wurde im Zustand $\mathbb{E}_3^+$ fixiert und die Parameter alle gleich belassen, nur die *unmanaged Redundancy* wurde kontinuierlich von 0 bis 1 erhöht (X-Achse). Die Auswirkungen an *ARI* (Y-Achse) durch die neu entstandenen *eFP* sind dabei deutlich sichtbar: während bei einer kleinen *unmanaged Redundancy* sich die zusätzlichen *eFP* im einstelligen Bereich bewegen, steigt die Kurve im späteren Verlauf stärker an. Man kann ebenfalls erkennen, dass sich im gesamten Verlauf die *Corrective Maintenance* niemals durchsetzen kann und hauptsächlich die *Preventive Maintenance* den Aufwand nach *ARI* beeinflusst. Dabei verläuft die Kurve für *Corrective Maintenance* tatsächlich wie gewollt, nur ist der resultierende Wert im Vergleich zu den anderen beiden relativ klein. Das liegt daran, dass die Nutzraten der Datenbestände im Vergleich zu dem Gesamtsystem sehr klein sind. Würde das komplette System voll und ganz von beiden Datenbeständen abhängen, wäre die y_c-Kurve bei Anstieg der $\overline{Rd}$ auch ersichtlich, so aber bewegt sie sich in Bereichen $0 < y_c < 0.1$. Dass die *Corrective Maintenance* nicht relevant ist, widerlegt allerdings eine spätere Evolution des Systems: durch eine Erweiterung von einigen neuen Elementen mit mehr Beziehungen steigt auch die *Corrective Maintenance* so stark an (was noch später in Kapitel 6 „System Simulation“ aufgezeigt wird), dass der entsprechende Faktor $q_c = 0.5$ gewählt werden musste. Schlussendlich lässt sich anhand Abbildung 5.21b das Problem der exponentiellen Funktion der *Preventive Maintenance* zeigen: wenn $\overline{Rd} = 0$ ist, dann liegt dennoch ein erhöhter *ARI* im System an, was gegen die Definition des *ARI* verstoßen würde, was bedeutet, dass die Metrik nicht für $\overline{Rd} = 0$ definiert ist. Sollte einmal das dabei zugrundeliegende $N(d_j) = 0$ sein (also ein Datenbestand im unbenutzten Zustand), gilt die Formel jedoch weiterhin wie gewollt. Die senkrechte gepunktete Linie in 5.21b stellt dabei den Punkt $\overline{Rd} = 0.08$ aus dem System 5.21a dar, was sich schlussendlich auf einen *ARI* von 2.0629 beläuft. Die Relativierung der Unternehmensfaktoren $q_a = 0.5$ und $q_c = 0.5$ ist dabei bereits (in den Kurven sowie der Berechnung) beachtet. Warum diese entsprechend gewählt wurden, wird in späteren Kapiteln genauer erläutert. Die genaue Berechnung ist in dem Beispiel aufgeführt, jedoch mit diversen Abkürzungen, weil sich viele Faktoren auf 0 berechnen und dadurch viele Teile der Metrik entfallen. Ebenfalls wurden alle Vektoren $\varphi \cdot \varphi'$ sowie Nutzungsverhalten $N(d_i)$



(a) Das Beispielsystem im Zustand $\mathbb{E}_3^+$



(b) Der Verlauf der Funktionswerte der Metriken bei $0 < Rd < 1$

$$y_a = ((0 \cdot \ldots) + 40 \cdot ((\ldots \cdot 0) + (1 \cdot [\approx 0.27] \cdot 0.08)) \cdot q_a \qquad \approx 0.864 \cdot q_a$$

$$y_c = ((0 \cdot \ldots) + 15 \cdot \frac{3-2}{3} \cdot (\sqrt{0.08} \cdot \frac{[\approx 30.1] \cdot [\approx 16.65]}{185^2}) + (40 \cdot 0 \cdot \ldots) \cdot q_c \qquad \approx 0.0207 \cdot q_c$$

$$y_p = (\frac{5500}{4250} \cdot ([\approx 16.65]^{0.08})) \cdot q_p \qquad \approx 1.6206 \cdot q_p$$

$$y(\mathbb{E}) = y_a \cdot q_a + y_c \cdot q_c + y_p \cdot q_p \qquad \approx 2.5053 \cdot q$$

$$y(\mathbb{E}) = y_a \cdot q_a + y_c \cdot q_c + y_p \cdot q_p = 0.864 \cdot 0.5 + 0.0207 \cdot 0.5 + 1.6206 \cdot 1.0 \qquad \approx 2.06295$$

Abbildung 5.21: Das Verhalten der Metriken bei Anstieg der Redundanz $\overline{Rd}$

bereits berechnet und in eckige Klammern $[\approx \varphi \cdot \varphi']$, beziehungsweise $[\approx N(d_i)]$, geschrieben, da ansonsten die Übersichtlichkeit nicht gegeben werden kann. Eine ausführliche Berechnung ist im Anhang A.1 aufgezeigt. Der nächste Schritt ist nun die Ermittlung des eigentlichen Wertes des Systems aus den System-*ARI*.

Die Ermittlung des Systemwertes k Ist der gesamte *ARI* für das System einmal bestimmt, ist dieser fix für den aktuellen Zustand. Eine neue Erweiterung erfordert jedoch eine erneute Berechnung des *ARI* für das komplette System. Betrachtet man alle einzelnen, sich für jeden Zustand des Systems von $\mathbb{E}_0^+$ bis $\mathbb{E}_x^+$ ergebenden Gesamt-*ARI*, dann erhält man eine Kette an Werten, in Abhängigkeit der Erweiterung des Systems. In jedem dieser Zustände kommt nun $y(\mathbb{E})$ zum Tragen, sodass jederzeit der „Aufwand“ des Gesamtsystems in *FP* gemessen werden kann und gleichzeitig der effektive Aufwand in *eFP* durch $eFP = FP + y(\mathbb{E})$ berechnet werden kann. Für das Beispiel in 5.21 ist der normale Aufwand für die reine Funktionalität im Zustand $\mathbb{E}_3^+$ bei 185 FP, der tatsächliche Aufwand berechnet sich damit durch

$$eFP = FP + y(\mathbb{E}) = 185 + 2.06295 = 187.06295$$

Setzt man beide Aufwände nun in ein direktes Verhältnis, kann man erkennen, inwieweit das System an Wert verloren hat. Dieses Verhältnis, die hier genannte „Wert-Formel“, ergibt sich demnach durch

$$k = \frac{FP(\mathbb{E})}{eFP(\mathbb{E})} = \frac{FP(\mathbb{E})}{FP(\mathbb{E}) + y(\mathbb{E})}$$

Man beachte, dass die „Wert-Formel“ für das System $\mathbb{E}$ definiert wird und die *FP* und *eFP* immer auf dieses Gesamtsystem bezogen werden, nicht auf eine einzelne Erweiterung, weswegen der Zustand der aktuellen Erweiterung fortan am Ende in eckigen Klammern gesetzt wird. Da in diesem Verhältnis der Zähler immer gleichzeitig auch komplett in den Nenner einfließt und $y(\mathbb{E}) \geq 0$ ebenfalls immer gilt, besitzt k jederzeit einen Wert zwischen 0 und 1. In einem System, welches aus keinerlei *unmanaged Redundancy* besteht, wird jede Metrik an bestimmten Punkten so belegt, dass deren Resultate jeweils 0 ergeben und damit der Wert des Systems stetig bei 1 verbleibt. Sobald eine der Metriken ein Resultat für *ARI* mit größer 0 ergibt, sinkt k kontinuierlich. Für das Beispielsystem in Abbildung 5.21 ergibt sich damit ein Systemwert k, beziehungsweise ein Verfall des Systems, von

$$k = \frac{185}{185 + 2.06295} \approx 0.98897\ [\mathbb{E}_3^+]$$

was soviel bedeutet, dass das System ungefähr 1.1% an Wert verloren hat, verursacht durch die eine *unmanaged Redundancy* zwischen zwei Datenbeständen. Dieser Wert ist dafür geeignet, den Verfall des Systems zu beobachten und dessen maximales und genutztes Potenzial sichtbar zu machen. Ebenfalls eignet sich dieser Wert durch seinen relativen Bezug dafür, das eigene System mit einem anderen zu vergleichen, da der Systemwert k immer durch das Verhältnis vom normalen und effektiven Aufwand gebildet wird. Um die genaue Bedeutung dieses Wertes für die *Agility* beurteilen zu können, müssen noch zwei letzte Schritte vorgenommen werden.

Die Kosten in TtM und DevC Neben dem Systemwert kann man den *ARI* auch als feste Konstante bestimmen, indem die überzähligen *eFP* eines Systems zurück in *TtM* und *DevC* umgerechnet werden. Dafür sind unternehmensspezifische Faktoren notwendig, da die folgenden beiden Fragen beantwortet werden müssen:

1. Wie viel Zeit benötigt ein Entwickler pro *FP*?

2. Was kostet eine Arbeitsstunde des Entwicklers?

Bei der ersten Frage ist es wichtig zu beachten, dass dabei nicht nur der Entwickler selbst eingeschätzt wird, sondern auch dessen Interaktionszeit mit allen notwendig beteiligten Steakholdern, wie Analysten, Designer und andere Entwickler. Die zweite Frage zielt ebenfalls auf mehr ab als das reine Gehalt des Entwicklers. Es spielt hierbei eine große Rolle, was die zugehörige Technik sowie Drittposten (zum Beispiel neue Anleihen von Cloud-Speicherplatz) kosten. Ebenso sollte der Aufwand der anderen daran beteiligten Personen einbezogen werden, wenn der Entwickler zum Beispiel mit einem Tester zusammenarbeiten muss. Hat man beide Fragen ausführlich und unternehmensspezifisch beantwortet, kann man mit der Systemanalyse fortsetzen. Dabei ergeben sich aus den empirisch ermittelten Werten für „Zeit pro *FP*“ und „Geld pro Stunde“ zusammen mit den aus den Metriken erhaltenen *eFP* die beiden Fixwerte für *TtM* und *DevC*. Dass *Function Points* zurückübersetzt werden können, ist durch die inhaltsbasierte Natur der *FP*-Metrik gegeben. Es ist das, was die *FP* ausmachen, sodass mit ihnen bereits in frühen Phasen der Projektentwicklung die Kosten abgeschätzt werden können. Es gibt dadurch eine sinnvolle Korrelation zwischen den *FP* und dem Aufwand durch *TtM* und *DevC*. Und dadurch, dass *eFP* von den normalen *FP* abgeleitet sind, ergibt sich auch der Zusammenhang mit dem *ARI*, welcher durch die Metriken berechnet wird. Man beachte hierbei, dass die berechneten Tage für *TtM* als Personentage zählen, welche normalerweise im Unternehmen nicht von einem Entwickler allein verursacht werden, sondern sich meist auf viele Entwickler und beteiligte Steakholder aufteilt, sodass 5 *TtM*-Personentage als 5-Mann-Team innerhalb eines Tages erledigt werden können. Deswegen sollte man sich nicht von sehr hohen berechneten *TtM*-Werten beirren lassen. Dagegen sind *DevC* gleichbleibend und „real“, da es unabhängig davon ist, ob nur eine Person alle Kosten verursacht oder mehrere Personen sich die Kosten teilen, denn in Summe bleibt damit die *DevC* gleich.

Für das Beispiel in 5.21 erhält man, wie oben beschrieben, durch die Wert-Formel einen System-Wertverfall von $\approx 0.98897\ [\mathbb{E}_3^+]$. Der für Reparaturen notwendige Mehraufwand *ARI* beträgt dabei 2.06295 *eFP*. Nimmt man an, dass ein Entwickler zwei Tage pro *FP* benötigt und ein Entwickler das Unternehmen im Schnitt 50 € pro Arbeitsstunde kostet, erhält man durch die Rückrechnung des *ARI* einen Zuwachs der *TtM* von ≈ 4.1 Tagen, was dadurch ebenfalls die *DevC* um ≈ 1650 € erhöht (bei acht Arbeitsstunden pro Tag). Das erscheint in Anbetracht der normalen *FP* von 185 sehr wenig, aber man muss hierbei bedenken, dass die 4.1 Tage und 1650 € nur durch eine Redundanz von $\overline{Rd} = 0.08$ entstanden sind, was definitiv vermeidbar gewesen wäre. Zudem sind die nicht-technischen Folgen wie Systemausfall durch die Metriken nicht mit erfasst, welche jedoch die Unkosten ebenfalls weiter erhöhen können. Bei sehr großen Systemen, welche nicht nur zwei sondern Hunderte von Datenbeständen enthalten, summiert sich die darin befindliche *unmanaged Redundancy* auf ein Vielfaches von dem hier errechneten, was dann durchaus ernstere wirtschaftliche Überlegungen in der System-Entwicklung auslösen kann. Die Frage nach dem Extremfall (was passiert bei $\overline{Rd} \approx 1.0$) im Bezug auf Abbildung 5.21b hat dennoch eine Berechtigung, da ein *ARI* mit 40+ *eFP* bei insgesamt 185 „normalen“ FP infrage gestellt werden kann. Unstrittig ist, dass die Synchronisation und Fehleranalyse für die komplette Kopie eines Datenbestandes sehr aufwendig ist, jedoch ist bislang ungewiss, ob so viel *eFP* praxisrelevant sind und was genau letzten Endes durch die Unternehmensfaktoren q eingegrenzt werden kann.

Die Agility des Systems Zum Schluss wird der letzte Wert aus der gesamten Rechnung ermittelt, die *Agility* des Systems. Auch für die *Agility* gibt es eine Metrik, den sogenannten *Agility Value AV*, entnommen aus Furrer [19]:

$$AV = \frac{(\sum Size_i)^2}{\sum TtM_i \cdot \sum \frac{DevC_i}{1000}}$$

Damit ergibt sich der AV durch das Verhältnis der quadratischen Größe zu der Summe aus TtM (in Tagen) und $DevC$ (in tausend €). Die $Size$ wird dabei in FP oder UCP gezählt, also genau das, womit in dieser Arbeit gehandhabt wird. Die Werte für TtM und $DevC$ stehen nun vor und nach der Architekturverletzung fest und somit sind alle notwendigen Werte ermittelt. Der AV des Beispiel-Systems ohne Architektur-Verletzung ergibt dabei:

$$AV_o = \frac{(185)^2}{(185 \cdot 2) \cdot (\frac{185 \cdot 2 \cdot 8 \cdot 50}{1000})} \approx 0.625$$

Zu beachten ist, dass TtM aus dem empirischen Wert an zwei Tagen pro FP und die $DevC$ aus der Anzahl der Arbeitsstunden mit 50 € pro Stunde ermittelt wurden. Die gleiche Formel ergibt für den AV des Systems mit Architektur-Verletzung:

$$AV_m = \frac{(185)^2}{(188.734 \cdot 2) \cdot (\frac{188.734 \cdot 2 \cdot 8 \cdot 50}{1000})} \approx 0.60$$

Man kann erkennen, dass der Wert für *Agility* um 0.025 gesunken ist. Was das genau bedeutet, muss mit dem Unternehmen selbst untersucht werden, da der AV ein reiner Vergleichswert ist und nicht absolut auf etwas bezogen werden kann. Dabei ist der AV-Wert durchaus monetär zu verstehen. Ein Verlust von 0.025 AV führt zwangsweise dazu, dass mehr Geld und Zeit pro FP investiert werden muss. Insgesamt kann man davon ausgehen, dass eine niedrige *Agility* früher oder später dem System mehr schadet, als wenn man stetig die *Agility* (und die *Resilience* wie auch den *Business Value*) hoch hält, wie es in Kapitel 2.4.2 „Zukunftsfähige Architektur: Agility, Resilience und der Business Value“ beschrieben wurde.

Das Ende des Nachweises Diese Wert-Formel sowie die Rückrechnung auf *TtM*, *DevC* und *Agility* bildet nun das Resultat der Quantifizierung in diesem Nachweis. Damit ist die Hypothese mit der in diesem Nachweis erbrachten Theorie dahingehend bestätigt, dass es einen kausalen und quantitativen Zusammenhang zwischen dem Wert des Systems (im Bezug auf dessen Evolutionsfähigkeit) und dessen Architekturprinzipien-Treue (hier die Datenredundanz) gibt. Die beiden folgenden Abschnitte werden noch näher auf die Dynamik des Modells (Darstellung und Messbarkeit) und die in dem Modell enthaltenen Fehler und Toleranzbereiche eingehen. Doch der Nachweis selbst, dass man den kausalen Zusammenhang aufweisen und quantifizieren kann, wird hiermit als erbracht angesehen.

5.5 Dynamisches Verhalten

Dieser Abschnitt dient dazu, einmal das zeitliche Verhalten des Systemwertes zu verfolgen. Trägt man zu jeder Erweiterung den ausgerechneten Gesamt-ARI $y(\mathbb{E})$ ab, erhält man die weiter oben angesprochene Folge von Werten. Durch eine kontinuierliche Messung des tatsächlichen Aufwandes des Systems zu jedem Zeitpunkt einer Erweiterung erhält man die Werte-Kette, welche eine Art zeitlichen Verlauf darstellt.

$\mathbb{E}_x^+$	init	1	2	3	4	5	6	7	8	9
FP	110	115	125	133	139	142	165	168	165	161
eFP	110.0	115.0	128.2	136.4	143.2	144.9	170.3	175.2	171.2	166.6
$\boldsymbol{k}$	1.0000	1.0000	0.9750	0.9750	0.9706	0.9799	0.9688	0.9589	0.9637	0.9663

$\mathbb{E}_x^+$	10	11	12	13	14	15	16	17	18	19
FP	165	171	172	172	180	185	188	190	191	198
eFP	168.7	175.5	177.0	174.3	189.1	192.1	195.0	195.7	197.9	208.3
$\boldsymbol{k}$	0.9780	0.9743	0.9717	0.9868	0.9518	0.9630	0.9641	0.9708	0.9651	0.9505

Tabelle 5.4: Fiktive Werte eines Systems

Nimmt man einen stetigen Zuwachs und Abbau an „echten“ FP durch Erweiterungen $\mathbb{E}_x^+$ im System an, erhält man die folgende Grenzwertbetrachtung

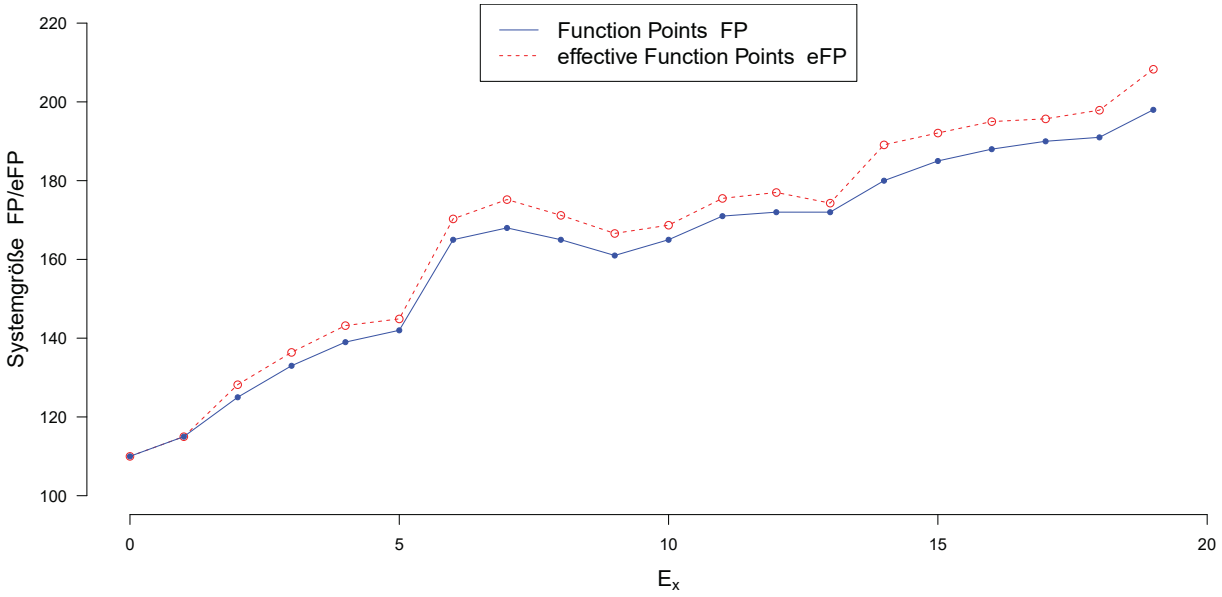
$$\lim_{FP(\mathbb{E})\to\infty} \frac{FP(\mathbb{E})}{FP(\mathbb{E}) + y(\mathbb{E})} = \frac{\infty}{\infty + c} = 1 \text{ wenn } y(\mathbb{E}) \text{ konstant}$$

Das bedeutet, sobald eine *unmanaged Redundancy* erkannt worden ist und dadurch ein Schaden in Form von ARI enthalten ist, tendiert das System anschließend dennoch gegen 1, da neue Erweiterungen *ohne Fehler durch Datenredundanz-Verletzung* den Wert des Systems wieder anheben können, auch wenn dadurch das System nie wieder ganz ideal wird, solange sich diese Prinzipien-Verletzung im System befindet.

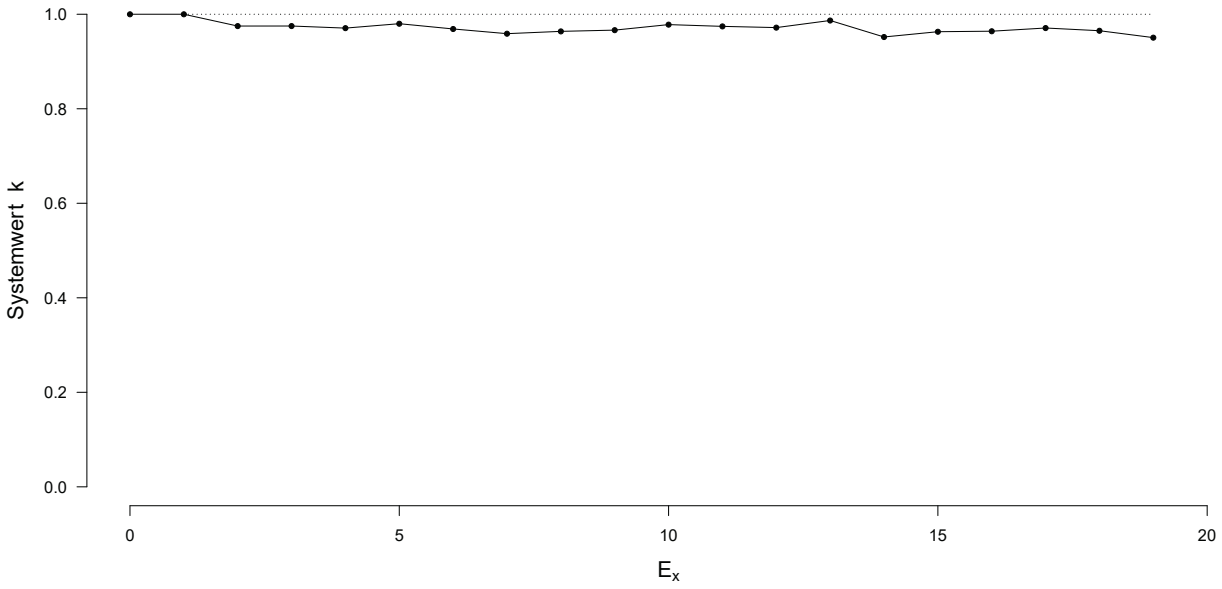
Ein Beispiel einer Systemwert-Veränderung ist in Tabelle 5.4 sowie den dazugehörigen Kurven in Abbildung 5.22 angegeben. Man kann erkennen, wie sich der Systemwert in der Tabelle vom Zustand $\mathbb{E}_0^+$ (dem `init`-System) bis hin zur 19. Erweiterung stetig verändert. Während in den ersten beiden Zuständen das System noch ideal ist, das heißt frei von *unmanaged* Datenredundanz, und damit einen k-Wert von 1.0 besitzt, sinkt dieser ab der dritten Erweiterung auf $k < 1$ ab. Wie man ebenfalls in diesem Beispiel erkennen kann, müssen FP und eFP nicht zwangsweise proportional zusammenhängen. Ein Beispiel hierfür ist die in Tabelle 5.4 vorgenommene Erweiterung $\mathbb{E}_{13}^+$, welche die gesamte System-Funktionalität nicht ändert, sondern hauptsächlich Datenredundanz entfernt, weswegen der eFP-Wert des Systems von 177 auf 174.3 sinkt und damit eine Steigerung von k verursacht wird, wie ebenfalls in den Kurven von Abbildung 5.22a und Abbildung 5.22b bei $x = 13$ ersichtlich ist. Man beachte, dass eine neue Erweiterung niemals „echt leer“ sein darf, also immer eine nicht-leere Menge an Elementen vorhanden sein muss. Diese Elemente können durchaus bereits vorhanden gewesen sein und durch die Erweiterung nur ihre Eigenschaften verändert haben. Damit könnten theoretisch auch solche Erweiterungen auftreten, welche keine neuen FP dem System hinzufügen sondern lediglich strukturelle Anpassungen an der bestehenden Komponenten-Struktur vornehmen.

Eine der Kernfragen bei der Analyse des dynamischen Verhaltens ist, wie sich das System durch die Metriken insgesamt ändert. Dafür gibt es zwei verschiedene Möglichkeiten der Betrachtung des Systems und dessen Dynamik:

1. Welche Dynamik weist k auf, wenn man das System in einer Erweiterung fixiert und nur die *unmanaged Redundancy* verändert?



(a) Verlauf der $FP(\mathbb{E})$ und $eFP(\mathbb{E})$ in Abhängigkeit von $\mathbb{E}_x^+$



(b) Verlauf des Systemwertes k in Abhängigkeit von $\mathbb{E}_x^+$

Abbildung 5.22: Verlauf des Systemwertes k

Hier werden auf Grundlage der fiktiven Werte aus Tabelle 5.4) zwei Diagramme gebildet: ersteres beschreibt visuell das System, wenn es im Laufe seiner Erweiterungen mehr *Technical Debt* in Form von *eFP* hinzu bekommt. Das zweite Diagramm stellt den Wertverlust des Systems dar, indem der Systemwert k für jeden Zeitpunkt einer Erweiterung über *FP* und *eFP* berechnet wurde. Dies repräsentiert sehr gut den langsamen Verfall des Systems beim Einbau neuer *Technical Debt*.

2. Welche Dynamik weist k auf, wenn das System stetig erweitert wird und neue *unmanaged Redundancy*'s hinzugefügt werden, diese jedoch fest sind?

Für die erste Frage konnte man in Abbildung 5.21 bereits erkennen, welches Verhalten die Metriken an sich aufweisen. Die Summenfunktion aus allen drei Metriken (die schwarze Kurve in Abbildung 5.21b) ergibt sich direkt aus $y(\mathbb{E}) = y_a + y_c + y_p$. Fixiert man das System in einem Zustand, dann verhält sich jede Redundanz wie dort abgebildet, da kein Faktor (ausgenommen von $r(\lambda)$) die Charakteristik der Kurven verändern, sondern maximal in der Höhe relativieren kann. Erhöht man in dem fixierten Systemzustand eine Redundanz, verringert sich der Systemwert entsprechend der schwarzen Kurve, anfangs mit einem langsam ansteigenden Verfall und bei höherer *unmanaged Redundancy* einer Steigerung dieses Anstiegs. Sollte sich durch den Kontext des Systems anfangs die *Corrective Maintenance* bemerkbar machen können, wird ebenfalls am Anfang der Verfall schneller vonstatten gehen, bedingt durch die Auswirkungen der Wurzelfunktion der Metrik. Dieses Verhalten kann auf jedes System übertragen werden, es bleibt dann nur noch die Frage nach der Relevanz (beziehungsweise der Vernachlässigbarkeit) des *ARI*, welche Höhe der endgültige Wert für *eFP* annimmt.

Die zweite Frage zielt auf das dynamische Verhalten, wenn das System wie in Beispiel 5.22 über die Zeit erweitert wird. Hierbei gibt es keine Generallösung, wie bei der ersten Frage nach der Dynamik. Hier kommt es zum Beispiel auf die einzelnen Redundanzen an, welche eingebaut werden, auf den Inhalt der Erweiterungen und dessen *FP* oder den Aufbau des Systems (inwieweit Komponenten die Datenbestände nutzen). Zwar kann man auch hierbei wieder die Summenfunktion annehmen, da mehrere Summenfunktionen sich zur selbigen addieren und die Charakteristik beibehalten, jedoch ist es hierbei sehr viel wichtiger, welche Redundanz sich in welcher Höhe einbringt, da zum Beispiel die *Corrective Maintenance* einer Komponente c_i die *Preventive Maintenance* von einer anderen Komponente c_j komplett überdecken kann. Dadurch ist die einzige Aussage, die über den zeitlichen Verlauf und die Dynamik möglich ist, jene, dass es mit neuen *unmanaged Redundancies* auch einen erhöhten Verfall von k des Systems geben wird, was auf die Metrik-Eigenschaft der *Monotonicity* zurückzuführen ist. Dadurch gibt es leider keine Möglichkeit, eine generelle Funktion zu finden, welche zum Beispiel den Verlauf des k in Abbildung 5.22b sinnvoll voraussagen könnte, da einfach zu viele Unbekannte in die Gesamtrechnung des k einfließen. Es wäre möglich, ein Voraussage-Modell zu entwerfen, ähnlich den Modellen für die Einschätzung von *Reliability* von Software [16], jedoch ist dies eine andere Thematik und keine grundlegende Eigenschaft von k.

5.6 Fehleranalyse und Toleranzen

Da nun alle wichtigen Bereiche abgedeckt sind, die Metriken beschrieben wurden, die Theorie erläutert wurde und damit die Hypothese belegt wurde, ist es noch wichtig, eventuelle Fehler, Toleranzbereiche und nicht beachtete Faktoren aufzuführen, um falschen Annahmen zuvorzukommen und zukünftige Arbeiten dahingehend zu sensibilisieren. Ebenfalls werden hier einige Probleme aufgezeigt, welche mit dem Modell und der gesamten Theorie einhergehen.

Der unermessliche Wert von Architektur Geht man die gesamte Kette an kausalen Zusammenhängen noch einmal durch, ist der erste Punkt, an dem Toleranzen hinsichtlich der Schlussfolgerungen entstehen können, jener, inwieweit man eine Architektur eines Systems fassen kann. Es gibt viele Modelle und Sprachen, welche Architektur beschreiben können, aber den echten Wert dahinter, das Gesamtbild des Systems ist damit meist nicht darstellbar. In Furrer [19]

wurde diese Thematik mit dem bekannten Satz „Das Ganze ist mehr als die Summe aller Teile.“ beschrieben. Diese Wertschöpfung im übergeordneten Sinne ist auch in anderen Themengebieten wiederzufinden: ein Gemälde, welches mehr Bedeutung beinhaltet als reine Farbkomposition, ein gut durchdachtes Gebäude, welches nicht nur die Funktionen einer Unterkunft erfüllt, oder eben ein Softwaresystem, welches so gut gebaut wurde, dass es einen Mehrwert gegenüber der initialen und reinen Funktionalität besitzt. Dieser Mehrwert von Software kann durch viele Prozesse geschaffen werden, jedoch ist die Architektur ein prominenter Vertreter dieser Prozesse. Aber so gut wie man sich den Mehrwert vorstellen kann, kann man ihn nicht wirklich ermessen. Was bei Gemälden durch Kunst-Schätzer vorgenommen wird, ist bei Softwarearchitektur undefiniert. Man kann es möglicherweise am wirtschaftlichen Durchbruch oder Konkurs einer Firma erkennen, ob der Mehrwert vorhanden war, aber ein geeignetes Tool, welches den Wert der Architektur-Leistung einschätzt, ist nicht vorhanden. Doch so ein Ansatz ist erforderlich, wenn man den Wert einer Architektur einordnen möchte. Der Ansatz mit dem Mehrwert durch hohe *Agility* und *Resilience* (beziehungsweise die Schätzung über *TtM* und *DevC*), welcher in dieser Arbeit gewählt wurde, basiert letzten Endes auf den in Murer *et al.* [41] und Furrer [19] genannten Ansätzen und Vorgängen. Diese beiden Arbeiten sind aus der Praxis entstanden und damit als verlässlich einzustufen, jedoch nicht als Generallösung für eine Grundlage zur Erfassung des Mehrwertes von Architektur anzusehen. Darauf aufbauend wurden hier die drei Architektur-Bereiche von *Adaptive*, *Corrective* und *Preventive Maintenance* als Wert-Grundlage benutzt. Auf dieser Basis setzt diese Arbeit auf und bildet eine kausal nachvollziehbare Kette auf Grundlage geeignet erscheinender Fakten oder Phänomen. Damit ist leider keine Garantie, Einzigartigkeit oder Perfektion in diesem Bereich gegeben.

Unbestimmtheit von Technical Debt und Architecture Erosion Das Messen der beiden Phänomene *Technical Debt* und *Architecture Erosion* ist in Fachkreisen bereits als schwierig bekannt, jedoch existieren bereits (empirische) Modelle, welche numerische Werte dafür errechnen können [12] [32] [42]. Die Datenredundanz als Teil von *Technical Debt* ist demzufolge ebenfalls ein schwer zu erfassendes Merkmal eines Systems, angefangen von dem realen Problem des Auffindens bis hin zu dessen Beseitigung. Davon abgesehen stellt sich die Frage, inwieweit man die *Technical Debt* bemessen kann, welche Aussagekraft der zugehörige Wert besitzt und in welcher Weise dieser zustande kommt. Wie in vorherigen Kapiteln beschrieben, basieren bisherige Modelle für *Technical Debt* meist auf Zähloperationen von Quellcode-Metriken, wie zum Beispiel die angesprochene *Cyclomatic Complexity* oder *Coupling* und *Cohesion*. Jedoch sind diese Metriken nur bedingt geeignet, den echten Charakter der *Technical Debt* zu erfassen, da der Verlust an *Agility* und *Resilience* nur eingeschränkt durch Quellcode-Metriken messbar ist. In dieser Arbeit wurde die Datenredundanz als *Technical Debt* derart gehandhabt, dass diese per Graph „ersichtlich gemacht“ wurde und mit einer speziell auf Datenredundanz zugeschnittenen Metrik versehen wurde. Auch diese Annahme, dass Datenredundanz im System durch Werte erfassbar ist, basiert auf theoretisch geeigneten Schlussfolgerungen. Ob die entstandene „formale“ Datenredundanz tatsächlich äquivalent der Praxis ist, kann auch infrage gestellt werden. Es ist die in dieser Arbeit favorisierte Vorgehensweise und erscheint sinnvoll um das Phänomen der Datenredundanz zu erfassen. Allerdings ist auch das nicht perfekt, da Datenredundanz so viele verschiedene Formen annehmen kann, dass nicht garantiert werden kann, dass der hier vorgestellte Ansatz alle Möglichkeiten von *unmanaged Redundancy* abdeckt.

Die Methodik der Methodik Der nächste größere Punkt, an dem gewisse Toleranzen nicht auszuschließen sind, ist die eigentliche Vorgehensweise, wie die Theorie dieser Arbeit belegt

wurde. Die in Kapitel 3 „Methodik und Metriken“ vorgestellten Methoden sind nur bedingt speziell auf Datenredundanz zugeschnitten. Es wird beschrieben, inwieweit die Konzepte von „Darstellung“ und „Messbarkeit“ die erforderlichen kausalen Vorgänger für die in dieser Arbeit benutzten Methoden von *Graphen* und *Metriken* sind. Leider ist diese Sichtweise der Unterteilung nicht oft in der Literatur vertreten und dient dennoch als Basis für die hier genutzte Kette an logischen Zusammenhängen. Statt einem großen kausalen Schritt werden dadurch zwei kleinere vorgenommen, was zusätzlichen Spielraum für Fehlannahmen schaffen kann, wobei das Problem sehr wahrscheinlich nicht in einem Schritt lösbar gewesen wäre. Der Verlauf über einen angepassten Graph mit Systemeigenschaften, welcher wiederum selbst als Ausgangspunkt für die Bestimmung des Wertes von Architektur dient, ist sehr vermutlich anfälliger für falsche Schlüsse als zum Beispiel eine direkte Ableitung des Wertes aus dem System. Ob dies möglich wäre, ist jedoch eine andere Frage.

Der Graph der Dinge Wie bereits bei der Methodik geschildert, bildet der Graph in dem Nachweis eine große Rolle. An ihm wird die Darstellung des Nachweises vorgenommen, das System übernommen und analysiert, Werte und Eigenschaften annotiert sowie erste neue Metriken bezüglich der Datenredundanz eingeführt. Der Graph an sich ist der erste von den beiden Teilen im Nachweis, welche direkte Fehler aufweisen können. Konnte man bei Toleranzen in der Methodik und der Auslegung von *Technical Debt* noch diverse Spannweiten vernachlässigen, ist es bei der Darstellung (und der Messbarkeit) ein recht kritisches Thema, da hierbei keine Auslegungssachen mehr vorhanden sind sondern aufgrund von „Fakten“ logische Schlüsse gezogen werden müssen. Da der entstandene Graph auf Grundlage des Systems aufgebaut ist, kann es zum einen diverse Systemeigenschaften geben, welche nicht im Graphen vorhanden sind, zum anderen zu viele Eigenschaften vorkommen, welche eigentlich keine echte Auswirkung auf die Messbarkeit haben und diese am Ende nur verfälschen. Auch der Mittelweg ist möglich, dass zwar im Grunde sinnvolle Eigenschaften im Graphen eingebaut sind, jedoch anders als eigentlich nötig gehandhabt werden. Deswegen wurde bei der Erstellung des Graphen darauf geachtet, möglichst relevante Eigenschaften aufzunehmen und diese begründet in die Struktur des daraus resultierenden *labeled, annotated, directed Multigraph* einzubetten. Für eventuelle, bereits bekannte Toleranzen bezüglich der Darstellung wurde im Graphen, wenn möglich, hohe Variabilität angeboten, um im Nachhinein gewisse Elemente des Graphen abzuändern, sollte eine Annahme sich als ungerechtfertigt herausstellen.

Algorithmen und Metriken Nachdem davon ausgegangen wird, dass die Herleitung des Graphen einen soliden Grund für die Messbarkeit bieten kann, ist es auch die Messbarkeit selbst, welche Fehlannahmen und Toleranzen aufweisen kann. Zuerst wurden die in dieser Arbeit erstellten Graph-Metriken definiert. Sie beschreiben einen Zustand einer Eigenschaft des Systems, indem verschiedene Faktoren einbezogen werden. Falsche oder ungenaue Annahmen an Systemeigenschaften könnten falsche Schlussfolgerungen in den Graph-Metriken nach sich ziehen. Ein Beispiel wäre hierbei die *Rd*-Metrik, welche mit dem Konzept der „Gleichheit“ versucht, die *unmanaged Redundancy* darzustellen. Jedoch ist davon abzusehen, diese Metrik als einzige Alternative für Redundanz-Beschreibung anzusehen. Es muss sich erst herausstellen, ob die Metrik für „Gleichheit“ auch die Praxis geeignet darstellt. Der nächste Punkt sind die Wert-Metriken, welche wie die Graph-Metriken ebenfalls Toleranzen aufweisen könnten. Diese sind hauptsächlich aus Graph-Algorithmen entstanden und wurden anschließend in eine mathematische Form gebracht. Da auch hierbei unbedachte Phänomene vorhanden sein könnten, können die Wert-Metriken leider keine Perfektion bieten. Da die Wert-Metriken auf den Graph-Metriken

aufbauen, welche wiederum auf dem erstellten Graphen aufbauen, ist ein unbedachter Faktor im Graphen sehr wahrscheinlich auch in den Wert-Metriken spürbar. Dabei kann man sich jedoch der Modularität der einzelnen Schritte bedienen, um falsche Annahmen und vergessene Elemente in die einzelnen Abschnitte der Theorie einzupflegen.

Die Metriken und Algorithmen sind neben dem Graphen ebenfalls Teile in der Theorie, welche konkrete Fehler enthalten und damit „echt“ falsch sein könnten, was die Annahmen betrifft. Um das auszuschließen, wurde versucht, die Einflussfaktoren genau abzuwägen, nur das Nötigste einzubeziehen und damit logisch schlussfolgernd geeignete Metriken zu finden. Weiterhin sei hier erwähnt, dass der Autor dieser Arbeit kein Mathematiker ist und die Metriken, besonders die Wert-Metriken, durch Zähl-Algorithmen auf dem Graphen entstanden sind. Wie bekannt sein dürfte, kann ein Algorithmus oft in mathematische Funktionen umgesetzt werden, indem zum Beispiel eine `for`-Schleife in eine $\sum$-Funktion umgesetzt wird. Die hierbei entstandenen Formeln für die Metriken wurden mehrmals kontrolliert und deren Funktionsweise überprüft, jedoch kann eine perfekte Umsetzung der zugrunde liegenden Algorithmen hin zu den Formeln nicht garantiert werden. Auch kann nicht ausgeschlossen werden, dass mögliche Vereinfachungen der Formeln (zum Beispiel durch Umstellen oder Zusammenfassen) nicht wahrgenommen wurden. Da im Kontext zu der Erstellung der Metriken jeweils textlich beschrieben ist, worum es sich handelt und wie die einzelnen Parts der Formeln funktionieren sollen, kann ein eventuell vorhandener Fehler jedoch abgedeckt sein. Man kann davon ausgehen, dass die Formeln, beziehungsweise die Metriken, keine großen mathematischen Fehler aufweisen. Eher sind logische Fehler in den Metriken wahrscheinlich, wenn eine bestimmte Situation nicht bedacht wurde, zum Beispiel bestimmte Belegungen von Parametern mit Extremwerten. Dann bietet es sich an, die Wert-Metriken als Ausgangspunkt für Verbesserungen zu nutzen. Von den Haupt-Metriken abgesehen, können die hier genannten Vorfälle natürlich auch auf alle anderen mathematisch beschriebenen Textstellen zutreffen.

Das Ende der Kausalität Am Ende können die Schlussfolgerungen, welche aus dem Graphen und den Metriken gezogen werden, an sich falsch sein. Zwar wird nicht davon ausgegangen, jedoch auch nicht explizit ausgeschlossen. Die Theorie wird durch eine Kette an logischen Schlüssen gestützt, die den Nachweis bilden sollen. Dabei ist es schwierig, eine theoretische Schlussfolgerung aus einer anderen zu ziehen, da die stützenden Fakten nur zu Beginn vorhanden waren. Alles Weitere basiert entweder auf Annahmen oder empirischer Ermittlung, welche im eigentlichen Sinne auch Annahmen bilden. Man kann sicherlich von einem Punkt des Nachweises in verschiedene Richtungen weiter logisch vorgehen, weswegen der hier erstellte Nachweis der Theorie kein Alleinstellungsmerkmal aufweisen kann. Man kann aber davon ausgehen, dass die hier gezogenen Schlüsse rational glaubwürdig sind und zumindest sinnvolle Teilaspekte der Realität wiedergeben. Ein Beispiel ist hierfür die Entscheidung, den direkten Zusammenhang zwischen den drei Kategorien *Adaptive*, *Corrective* und *Preventive Maintenance* einzubauen, in der Form, dass sich die drei Algorithmen gegenseitig bedingen. Obwohl dies tatsächlich eine valide Schlussfolgerung gewesen wäre, wurde im Rahmen dieser Arbeit entschieden, dass „nur“ die unterschiedlichen Ausprägungen der einzelnen Metriken und dessen Summenfunktion (vorerst) als geeigneter Zusammenhang der drei Bereiche dienen kann. Hierbei kann man bei zukünftigen Arbeiten noch weitergehen, was die jetzige Annahme aber wahrscheinlich nicht falsch werden lässt.

Des Weiteren unterliegen alle neu eingeführten Konzepte der Annahme, dass diese auch sinnvoll die Realität widerspiegeln: das Konzept der *eFP*, der „Gleichheit“ von *Rd*-Beziehungen, der Beschreibung von *unmanaged Redundancy*, der Messung der *Agility* über eine Rückrechnung der *eFP* in *TtM* und *DevC*, der grundlegenden Beschreibung vom Wert der Architektur über die

drei oben genannten Bereiche der *Maintenance*, dem Abstrahieren eines Systems in dem erstellten Graphen, den Wert und „Verfall“ des Systems in Form von k und weiteres. Um die Theorie mit ihrer Hypothese nachzuweisen, wurden die in dieser Arbeit erstellten Konzepte genutzt, was eine gewisse Unsicherheit mit sich bringt, da bislang kein großes Feedback eingeholt und verarbeitet werden konnte, bedingt durch die Neuheit des Ansatzes in dieser Arbeit. Es wird die Aufgabe von zukünftigen Arbeiten sein, dass die hier beschriebene Theorie und der dazugehörige Nachweis bestätigt, widerlegt, erweitert oder verbessert wird. Doch das, was als Ziel dieser Arbeit galt, wurde dahingehend erreicht, dass ein möglicher Weg an kausalen und quantifizierbaren Zusammenhängen zwischen Architekturprinzipientreue und dem Wert des System aufgezeigt wurde.

6 System Simulation

Das folgende Kapitel soll ein Tool vorstellen, welches im Rahmen dieser Arbeit entworfen wurde. Dieses „Simulation Tool“ soll ein Demo-Werkzeug zu der in dieser Arbeit erstellten Theorie darstellen. Das bedeutet, dass damit keine Bestätigung der Theorie beobachtet werden soll, sondern lediglich Computerunterstützung genutzt wird, um bestimmtes Verhalten der Ausarbeitungen ersichtlich zu machen. Im Kern geht es dabei um die Frage, wie sich der Systemwert k verhält, wenn ein System kontinuierlich von `init` auf $\mathbb{E}_x^+$ gesteigert wird. Bisherige betrachtete Beispielsysteme wiesen maximal einen Zustand von $\mathbb{E}_4^+$ auf, jedoch soll ebenfalls die Situation betrachtet werden, wie sich k verändert, wenn mehrere tausend Erweiterungen hinzugefügt werden. Da die entsprechenden Metriken und Graphen sehr groß und in der Berechnung sehr aufwendig werden, wird das hier vorgestellte Tool genutzt, um solche Situationen zu vereinfachen. Dabei geht es nicht um eine Simulation von real existierenden Systemen, sondern um eine Simulation von der Theorie für eine hohe Anzahl an ansonsten nicht mehr per Hand berechenbaren Systemzuständen. Dabei ist dieses Kapitel wie folgt aufgeteilt. Nach einer kurzen Einführung in die Entwicklung werden kurz die Möglichkeiten und Funktionen des Tools aufgezeigt, wie diese implementiert wurden und weshalb bestimmte Entscheidungen während der Entwicklung getroffen wurden. Dabei geht es hauptsächlich um Verständnis der Sachlage, weniger um eine Dokumentation des Quellcodes oder Ähnliches. Im zweiten Teil wird ein Test-Szenario gewählt, entsprechende Simulationen durchgeführt und anschließend ausgewertet. Im Abschluss wird noch eine Gesamt-Auswertung vorgenommen, einmal bezüglich der Simulationsfähigkeit des Tools und einmal bezüglich der Theorie. Da dieses Tool nicht den Kern der Arbeit darstellt sondern vielmehr als Zusatz angesehen werden kann, wird das gesamte Kapitel nicht so umfangreich ausfallen wie der eigentliche Nachweis der Theorie.

6.1 Intention und Korrektheit

Dieser kurze Abschnitt soll nur noch einmal kurz verdeutlichen, was im Bezug auf das Tool wichtig ist. Insbesondere soll hier kurz erwähnt werden, was für Erwartungen an das Tool auszuschließen sind.

1. *Mathematische Korrektheit* Auch wenn dies besonders in dem vorangehenden theoretischen Teil dieser Arbeit betont wurde, kann es immer passieren, dass bestimmte Eigenheiten der Metriken nicht so umsetzbar sind, wie man es per Hand ausrechnen würde. Das betrifft vor Allem die Vorgehensweise der Berechnungen. Während man händisch die Metriken in einigen Situationen schneller erfassen kann, benötigt der zugehörige Algorithmus bestimmte Formulierungen, um zum Beispiel Situationen auszuschließen oder anders zu handhaben.

2. *Programm-technische Korrektheit* Das Tool wurde darauf ausgelegt, die Metriken, beziehungsweise deren Algorithmen, so umzusetzen, dass die mathematische Funktion erfüllt wird. Dabei kann es immer passieren, dass ein logischer Fehler in der Programmierung des Tools etwas Anderes verursacht als erwartet. Das Tool wurde daraufhin überprüft und die Ergebnisse in mehreren Szenarien durch händische Rechnung überprüft, jedoch sind damit

logische Fehler noch nicht ausgeschlossen. Es kann nach einer Fertigstellung von Software so gut wie nie ausgeschlossen werden, dass noch Fehler und Bugs vorhanden sind.

3. *Fehleranfälligkeit und Stabilität* Das Tool wurde darauf zugeschnitten, eine einfachere Erweiterung des Systems und dessen Berechnung zu automatisieren. Das bedeutet weiterhin, dass nur marginal auf fehlerhafte Eingaben oder unerwartete Situationen korrekt reagiert wird. Beispielsweise wird darauf vertraut, dass das Input-File korrekt ist, da bereits beim Laden des Tools mehrere Berechnungen vorgenommen werden, weswegen Fehler im Input-File zu Programmfehlfunktionen führen können. Wenn das Programm bestimmte Parameter vom User abfragt, dann wird darauf vertraut, dass diese korrekt und sinnvoll eingegeben werden. Wer das Programm zum Absturz bringen möchte, wird dies sicherlich auch schaffen.

4. *Benutzung* Die Handhabung des Programms ist minimal gehalten, da zum einen es kein aufwendiges Benutzerinterface benötigt, zum anderen um sich auf das Wesentliche des Tools zu konzentrieren, was die Erweiterung und Berechnung des Systems betrifft. Zudem wird das Tool dadurch einsatzfähiger, da ein Benutzerinterface eventuell besondere Ansprüche an die Plattform stellen würde.

5. *Performance und Technik* Das Programm ist in der Sprache *Java* geschrieben, welche durchaus nicht für Performance bekannt ist. Das Gute an *Java* ist die einfache Benutzung sowie die plattformübergreifende Funktionalität, sodass das Programm theoretisch auf verschiedensten Betriebssystemen laufen könnte. Abgesehen der mangelnden Performance von Java wird hier ausgeschlossen, die besten Lösungen für die jeweiligen Problematiken der Metriken gefunden und eingebaut zu haben. Es ist recht sicher, dass eingebaute Mehrfachiterationen und ungünstig gewählte Implementierungen die Performance des Programms senken können und werden. Dies tut dem Ganzen jedoch kein Abbruch, da das Tool nicht für schnelle sondern für richtige Berechnung geschrieben wurde.

6.2 Das Simulation Tool

Das Tool soll in diesem Kapitel kurz vorgestellt werden. Dabei soll hier keine Dokumentation dargestellt werden, sondern vielmehr die interne Funktionsweise nahegebracht und der Workflow erläutert werden.

6.2.1 Anforderungen und Spezifizierungen

Die Anforderungen an das Tool waren recht einfach zusammengefasst: „Das Tool soll eigenständig große Systeme erstellen und die Metriken daran berechnen“. Die Auslegung war dabei Teil dieser Arbeit und hatte im Kern folgende Anforderungen an das Resultat:

1. Ein System kann eingegeben werden.

2. Ein System wird nach den Metriken berechnet.

3. Das System kann nach Zufallsprinzip beliebig erweitert werden.

4. Das End-System kann jederzeit ausgegeben werden.

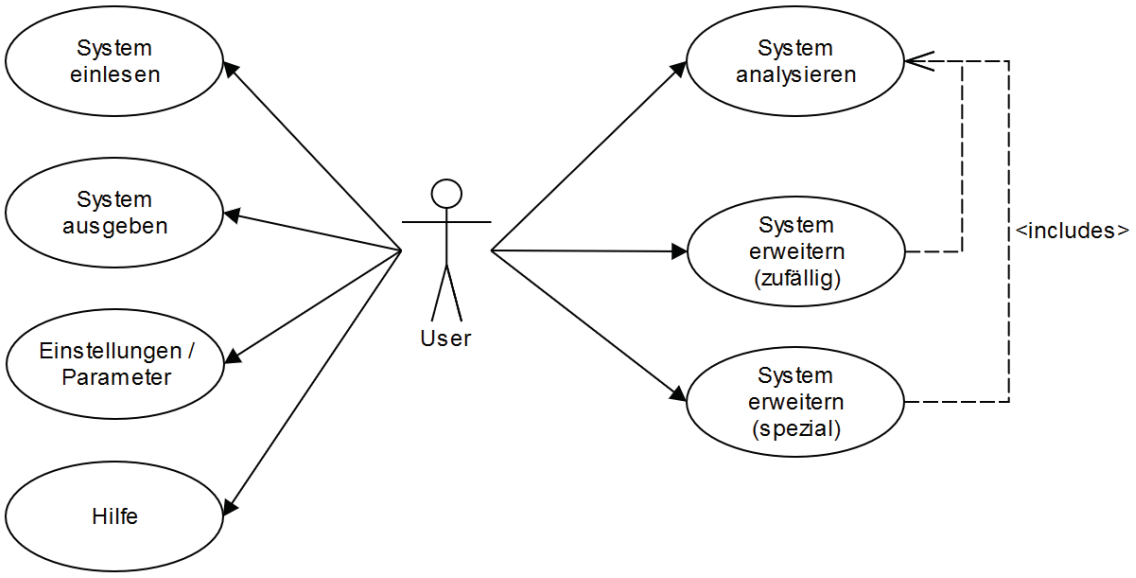


Abbildung 6.1: *Use-Cases* im Simulation-Tool

Dies sind die wenigen Anforderungen, die es benötigt, um die manuelle Berechnung der Metriken zu automatisieren. Um das Tool auch benutzbar zu machen, wurden im Vorfeld aus den Anforderungen gewisse *Use-Cases* abgeleitet. Durch neue Features sind auch neue *Use-Cases* entstanden. Das Resultat ergab dann folgende Menge, dargestellt als technisches UML-UseCase-Diagramm in Abbildung 6.1 und hier nochmals mit Kurzbeschreibung aufgelistet:

1. *System einlesen* macht nichts weiter, als ein System (definiert über ein Input-File) in das Tool einzulesen. Anschließend kann es dann analysiert und erweitert werden.

2. *System ausgeben* ist der Gegenpart zum Einlesen und schreibt das interne System aus dem Tool in ein Output-File.

3. *Einstellungen/Parameter* ist der Bereich, in dem der User diverse Parameter setzen kann, welche für die Berechnung (Analyse und Erweiterung) wichtig sind.

4. *Hilfe* ist eine kurze Auflistung von Informationen rund um das Tool.

5. *System analysieren* ist die erste Hauptfunktion und analysiert das interne (eingelesene) System nach den drei Wert-Metriken und gibt Statistiken aus, unter Anderem verschiedene Werte wie der Gesamt-ARI oder die zusätzliche TtM.

6. *System erweitern* ist der zweite Haupt-Part des Tools, hier wird das interne System durch Zufälligkeit erweitert. Dazu werden Parameter benutzt, welche der User definieren kann. Der Spezial-Fall bedeutet nur, dass statt vieler zufälliger Erweiterungen ganz konkrete Erweiterungen durch den Nutzer vorgenommen werden können, zum Beispiel eine Redundanz hinzufügen.

Insgesamt stand die Anforderung, ein Programm zu schaffen, welches die im theoretischen Part ausgearbeiteten Metriken und Funktionen automatisiert berechnet und das System beliebig (zufällig) erweitern kann. Bewusst wurden dabei Themen wie Benutzerinterface oder

bestmögliche Implementierung hinten angestellt. Das gibt insgesamt viel Spielraum für Weiterentwicklungen des Programms. Allerdings ist es so, wie es derzeit erstellt wurde, für die hier genannten Anforderungen und *Use-Cases* geeignet. Es entstand ein simples Konsolen-Programm, welches die nötigen Funktionen laut Anforderungen enthält.

6.2.2 Entwicklung und Funktionsweise

Die in dem theoretischen Part dieser Arbeit ausgearbeiteten Metriken wurden in Algorithmen übersetzt und in ein System eingebettet, das es erlaubt, mit Komponenten, Datenbeständen, Nutzungen und Redundanzen zu hantieren. Jedoch soll hier nicht weiter auf die Programmierung als Tätigkeit eingegangen werden, sondern vielmehr auf die erstellten internen Strukturen und zugehörigen Entscheidungen.

Graph-Darstellung Der wohl wichtigste Part bei der Entwicklung war die Darstellung des *labeled, directed, annotated Multigraphen*. Es gibt derzeit mehrere etablierte Möglichkeiten, einen Graphen in eine Struktur für Berechnungen zu bringen.

Die bekannteste Methode wird wohl eine *Adjazenzmatrix* [60] sein, welche sehr gut für Graphen mit Eigenschaften geeignet ist. In solch einer Matrix werden alle Knoten des Graphen jeweils als Spalte und Zeile angesehen, sodass sich eine gleichförmige Matrix ergibt. Der Inhalt der jeweiligen Zellen ergibt sich dann als Kanten-Beziehung zwischen zwei Knoten. Für den Graphen in dieser Arbeit würde dies formal wie folgt aussehen:

$$\mathbb{E}(\{\mathbb{C}, \mathbb{D}\}, \mathbb{X}) = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

mit $a_{m,n}$ als entweder Nutz-Beziehung φ zwischen c_m und d_n oder als Redundanz zwischen d_m und d_n. Dabei ist das a selbst eine Eigenschaft, beziehungsweise Annotation der Kante selbst. Die Gründe, warum diese Methode nicht gewählt wurde sind folgende:

- Die Matrix kann sehr schnell sehr groß werden, da für jeden weiteren Knoten eine Zeile und eine Spalte hinzugefügt wird.
- Meist ist es nur sinnvoll, pro Zelle eine Kanten-Eigenschaft zu speichern. Mehrere Eigenschaften erfordern einen höheren Aufwand für die Struktur.
- Durch verschiedene Typen von Knoten und Kanten ist es sehr umständlich, eine entsprechende Matrix zu erstellen, da bestimmte Eigenschaften nicht so einfach umgesetzt werden können, wie zum Beispiel, dass es keine Beziehung zwischen zwei Komponenten geben kann.
- Knoten-Eigenschaften müssten zusätzlich separat gespeichert werden, da die Zellen der Matrix nur Annotationen von Kanten bereit halten können.

Die nächste Möglichkeit, welche in Erwägung gezogen wurde, war die Nutzung eines vorgefertigten Graph-Datenbanksystems. Davon sind bereits einige etabliert, in Sadalage & Fowler [46] werden explizit die Systeme *Neo4J*, *Infinite Graph*, *OrientDB* und *FlockDB* genannt.

Der Vorteil an solchen Graph-Datenbanksystemen liegt darin, dass diese bereits vorgefertigte Strukturen anbieten, um einen Graphen abzuspeichern. Dabei ist bei den meisten davon die Größe und Verzweigung des Graphen nebensächlich. Intern werden die Graphen zwar unterschiedlich abgespeichert (zum Beispiel wieder über Adjazenzmatrizen oder andere zusammenhängende Strukturen), nach außen dem User gegenüber jedoch als kompletter Graph dargestellt. Die bekannten SQL-Mustersprachen aus relationalen Datenbanksystemen wie *DB2* oder *MySQL* sind hier nicht vorzufinden, stattdessen wird mit Funktionssyntax im eigenen Programm auf dem Graphen entlang propagiert [46]. Einige Beispiele (speziell für *Neo4J*) wären hierfür das Erstellen von Knoten, das Setzen von Eigenschaften, das Erstellen von Beziehungen, das Suchen von Knoten, sowie weiteren Manipulationen und Operationen auf Annotationen, Beziehungen und Knoten.

```
1  // Knoten erstellen
2  Node glados = graphDb.createNode();
3  Node cave = graphDb.createNode();
4
5  // Eigenschaften
6  glados.setProperty("name","GLaDOS");
7  cave.setProperty("name","Cave Johnson");
8
9  // Beziehungen (Kanten) erstellen
10 cave.createRelationship(glados,OWNER);
11
12 // Suchen
13 Node myNode = nodeIndex.get("name","GLaDOS").getSingle();
```

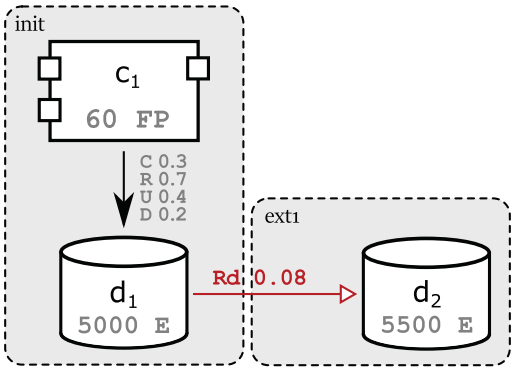
Quellcode 6.1: Operationen in *Neo4J*

Dass solche Graph-Datenbanksysteme (speziell *Neo4J*) für den operativen Einsatz geeignet sind, ist ebenfalls durch Datenbank-typische Features wie *Transaktionskapselung*, *Konsistenzsicherung*, *Hochverfügbarkeit*, hoher *Skalierbarkeit* und umfangreicher Anfragesprache (als Funktionen) sichergestellt [46]. Wie genau diese Datenbanksysteme intern arbeiten, ist für den User unerheblich, da das System bereits alle Bereiche von der Verwaltung bis zur Struktur abnimmt und sinnvolle Features für Graphen anbietet. Tatsächlich wäre eine solche Graph-Datenbank geeignet für die Nutzung des in dieser Arbeit erstellten Graphen gewesen. Man hat sich jedoch dagegen entschieden, da die Speicherung des Graphen sowie die angebotenen Features nicht im Vordergrund des Programms stehen, welches hier entwickelt wurde. Auch wurde bewusst auf einen umfangreichen Graph-Speicher verzichtet, um das Programm so klein und tailliert wie nur möglich zu halten, und dennoch die wirklich notwendigen Operationen anbieten zu können.

Letzten Endes wurde für die Speicherung des Graphen eine primitive Struktur ausgewählt. Insgesamt sind intern vier `LinkedLists` enthalten, jeweils eine für Komponenten, Datenbestände, Nutzbeziehungen und Redundanzen.

```
1 LinkedList<Comp> comps = new LinkedList<Comp>();
2 LinkedList<Data> datas = new LinkedList<Data>();
3 LinkedList<Use> uses = new LinkedList<Use>();
4 LinkedList<Red> reds = new LinkedList<Red>();
```

Quellcode 6.2: Die Speicherung aller Graph-Elemente




```
LinkedList<Comp> comps;
LinkedList<Data> datas;
LinkedList<Use> uses;
LinkedList<Red> reds;

// Eingabe des Graphen
comps.add(new Comp(1,0,60));
datas.add(new Data(1,0,5000));
datas.add(new Data(2,1,5500));
uses.add(new Use(1,1,1,0.3,0.7,0.4,0.2));
reds.add(new Red(1,1,2,RedType.STD,0.08));
```

Abbildung 6.2: Die Übertragung des Graph-Modells in die vierteilige Listenstruktur

Die Graph-Elemente selber sind einzelne Klassen, sodass am Ende ein Objekt einen der vier Typen darstellt und in einer der vier Listen gespeichert ist. Das ist dahingehend praktikabel, dass man mit einem Mal über alle Elemente einer Art iterieren kann. Das Vorgehen bei der Übertragung vom Graphen in die Listen ist dabei in Abbildung 6.2 aufgezeigt. Die Schwierigkeit besteht dann jedoch darin, die Verbindungen herzustellen. Sucht man eine spezielle Redundanz zwischen zwei Datenbeständen, ist die Frage, wo speichert man den Anfangs- und Endpunkt der Beziehung? Durch die Speicherung aller Redundanzen in einer eigenen Liste ist es notwendig, jeweils den Anfang und das Ende als Referenz zu den entsprechenden Datenbestand-Objekten abzuspeichern. Dies verursacht einen Mehraufwand beim Suchen von bestimmten Beziehung durch häufiges Iterieren über mehrere Listen, jedoch ist dies für den Zweck dieser Anwendung hinnehmbar.

Workflow Aufbauend auf die vier Speicher-Listen können die notwendigen Operationen abgebildet werden. Einfache `Util`-Funktionen lösen oft auftretende Probleme wie das Finden bestimmter Elemente nach ihrem Index oder das Lesen und Schreiben in In- und Output-Files.

Konkret sind die beiden Ankerpunkte im Ablauf des Programms die Funktionen `extend()` und `analyse()`. Beide Funktionen sind separat aufrufbar, wobei die Funktion folgendermaßen aussieht:

1. `analyse()` wertet das System nach dem *ARI* aus. Dafür berechnet die Funktion in drei Schritten alle Metriken und addiert die resultierenden Werte auf. Für jede der Metriken wird das System erneut als Ganzes analysiert, das bedeutet, dass immer alle Elemente in allen vier Listen einbezogen werden. Zudem werden hierbei andere Werte wie $\sigma'(c_i)$, $N(d_j)$ oder vererbte Redundanzen neu ausgewertet oder vorberechnet.

2. `extend()` erweitert das System im Sinne der hier beschriebenen Erweiterungen. Da es zu umständlich wäre, alle nötigen Elemente einzeln einzufügen, gibt es zwei verschiedene Modi: (a) den Zufallsmodus, welcher zufällig nach Normal-Verteilung verschiedene Elemente hinzufügt, und (b) den Einzelmodus, welcher ein ganz bestimmtes Element hinzufügt.

Bei `extend()` liegt der Fokus hauptsächlich auf der ersten Variante, um die Auswirkungen der Metriken bei besonders großen Systemen zu testen. Da dabei die Zufälligkeit (wie genau ein System erweitert wird) eine erhebliche Rolle spielt, wie sich die Evolution eines Systems äußert, reagieren auch die Metriken entsprechend auf diese vom Zufall gesteuerte Systemevolution. Eine ungenügend gewählte Zufälligkeit einer Evolution wird keine aussagekräftigen Werte durch

die Metriken erhalten und positiv dazu analog. Deswegen muss auf eine möglichst realistische Verteilung des Zufalls geachtet werden, welche Programm-intern mit einer *Gaußschen Normalverteilung* [64] realisiert wurde, da diese Verteilung vielen natur-, wirtschafts- und ingenieurwissenschaftlichen Vorgängen (beziehungsweise dem Verhalten von natürlichen Zufallsereignissen) nahe kommt, auch wenn durchaus alternative Modelle noch näher der Natur (oder zumindest bei Verteilungen in der Biologie) herankommen könnten [52]. Im Programm werden zum Beispiel die implementierten Zufalls-Parameter `add_items_mean` und `add_items_deviation` dafür benutzt, um eine Wahrscheinlichkeit nach Gauß zu simulieren, mit der eine tatsächliche Änderung am System während einer Erweiterung getätigt wird. Beide Werte sind dabei vom Nutzer vorher definierbar. Diese „natürliche“ Verteilung findet im Programm (und damit während der Simulation) bei so gut wie allen zufälligen Werten statt, immer dann, wenn zu entscheiden ist, wann, wo und wie ein Element des Systems behandelt wird.

Der allgemeine Ablauf einer Simulation – die Hauptaufgabe wofür das Programm geschrieben wurde – ist Folgender:

1. Ein Input-File wird definiert. Darin liegt das initiale System (`init`) mit allen notwendigen Elementen im CSV-Format vor. Ausführungen folgen im nächsten Abschnitt.

2. Die Parameter werden eingestellt (wenn nicht schon geschehen). Dazu zählen Parameter für alle Zufälle und allgemeinen Einstellungen (zum Beispiel die Kosten für Mitarbeiter und dessen Arbeitszeit für einen *FP*).

3. Ein `extend()` erweitert das System nach Zufall. Dafür werden die zuvor spezifizierten Parameter benutzt.

4. Optional wird nach jeder Erweiterung ein `analyse()` ausgeführt, ansonsten mindestens je ein `analyse()` am Anfang und Ende des Erweiterung-Prozesses.

5. Durch das `extend()` werden zwei Output-Files angelegt:
   a) eine Protokoll-Datei *output.txt* für alle vorgenommenen Erweiterungen, inklusive aller hinzugefügten Elemente sowie die Auswertung eines eventuellen `analyse()`, und
   b) die Statistiken aus den optionalen `analyse()`-Prozess nach jeder Erweiterung im CSV-Format in der Datei *output_table.csv* für spätere Weiterverarbeitung. Hinweis: beide Dateien werden bei erneutem `extend()` überschrieben.

6. Das fertig erweiterte System kann nun ebenfalls in eine Datei *output_system.txt* geschrieben werden, welche das gleiche Format wie das Input-File besitzt.

Die beiden *Use-Cases*, welche davon abweichen, sind entweder, das Input-System nur per `analyse()` zu messen (also auf das Erweitern zu verzichten), oder anstatt zufällige Erweiterungen nur ganz spezielle Elemente einzufügen, um bestimmte Auswirkungen zu testen. Das fertige Programm kann man in Abbildung 6.3 als UML-Entwurfsdiagramm betrachten, in welchem die wichtigsten Funktionen und Beziehungen dargestellt sind.

Aus Interesse und der Machbarkeit halber wurde das erstellte System ebenfalls als *labeled, directed, annotated Multigraph* in Abbildung 6.4 dargestellt. Man kann erkennen, dass keine *unmanaged* Datenredundanz vorliegt. Das liegt daran, dass das Programm recht klein ist, sodass die Datenbestände übersichtlich genug waren, um Duplikate auszuschließen. Das Einzige, was letztendlich existiert, sind noch synchronisierte Redundanzen (in Form von Referenzen), welche laut Annahme als *managed* gehandhabt werden. Weiterhin sind in der Abbildung die Werte für

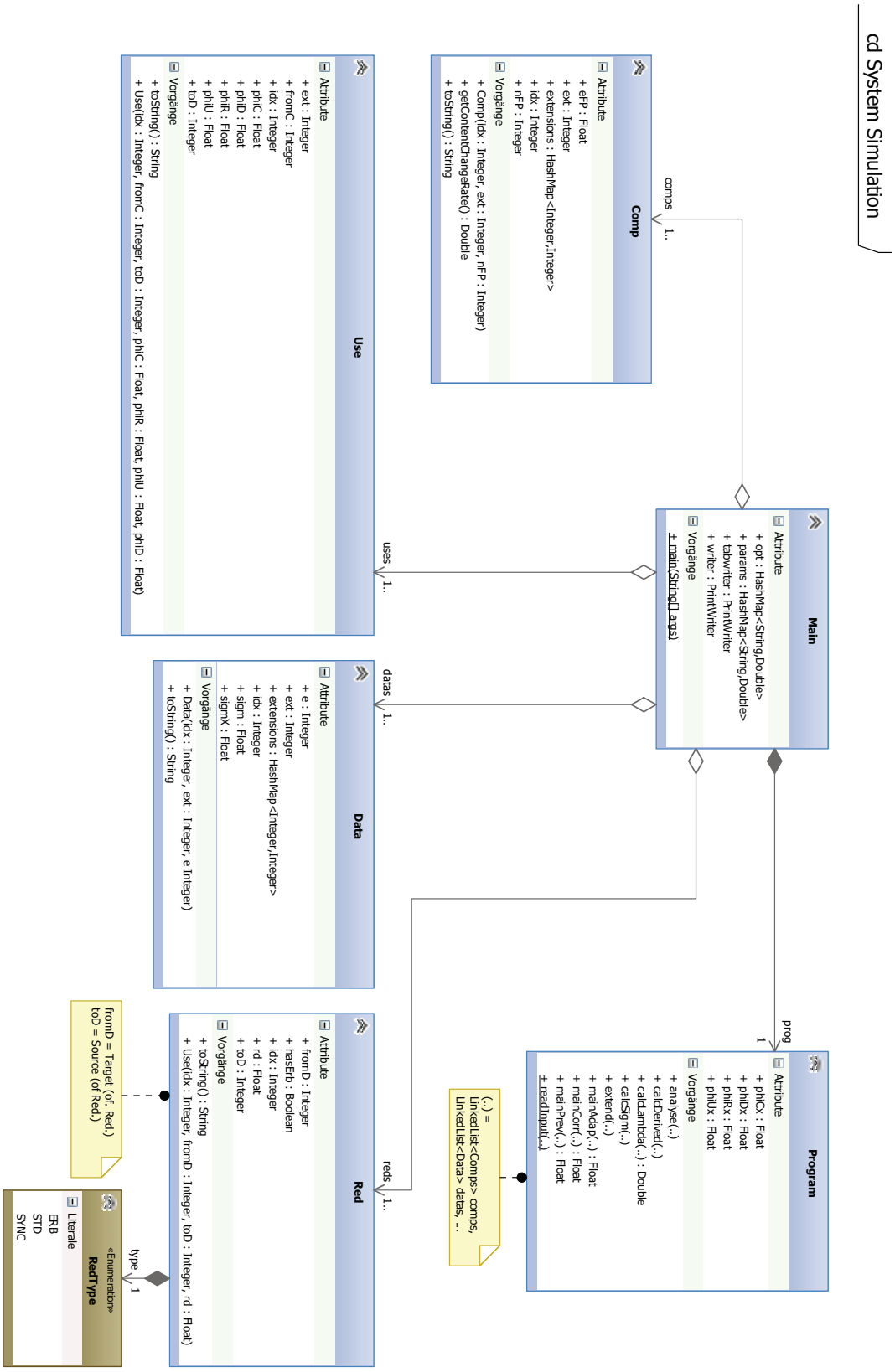


Abbildung 6.3: Das Programm als UML-Diagramm

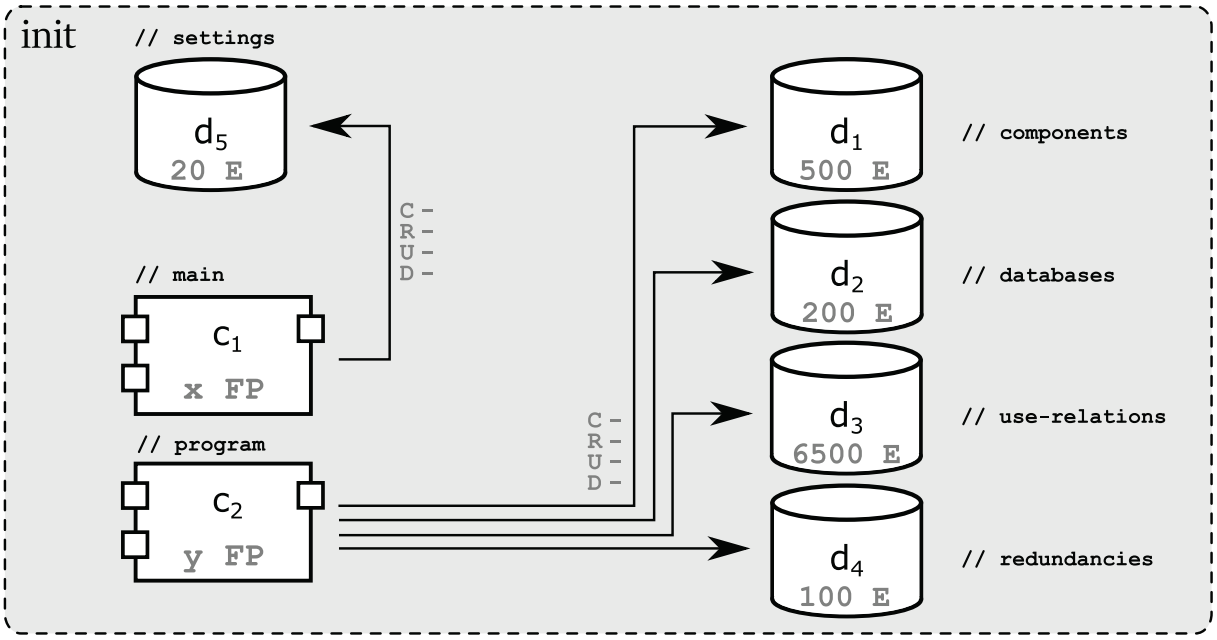


Abbildung 6.4: Das Programm als Graph dieser Arbeit

φ nicht angegeben und die *FP* nicht ausgerechnet, da durch die fehlende *unmanaged* Datenredundanz kein Schaden entstehen kann. Nur die *Entities* der Datenbestände sind angegeben, aber auch hier sind dies eher (realistische) Beispielwerte. Bei der Erstellung dieses Graphen wird ersichtlich, dass noch einige Fragen offen bleiben, wenn man tatsächlich versucht, reale Systeme auf das Modell abzubilden. Dieser Punkt kann in zukünftigen Arbeiten weiter ausgebaut werden.

6.2.3 Darstellung des Nachweises

Die Darstellung wird in dieser Arbeit von einem *labeled, directed, annotated Multigraph* übernommen. Dieser stellt das System dar. Das System, welches als Basis dienen soll, wird als Input-File *input.txt* noch vor Programmstart angelegt. Dieses kann zum Beispiel wie folgt aussehen:

```
#components: Idx,Ext,FP
C,1,0,20
C,2,1,110
C,3,2,15
C,4,3,40

#databases: Idx,Ext,E,*sigma,*sigma'
D,1,0,3000,0,0
D,2,2,5500,0,0

#uses: Idx,fromC,toD,phi_C,phi_R,phi_U,phi_D
U,1,1,1,0.1,0.3,0.3,0.7
U,2,2,1,0.0,0.0,0.4,0.2
U,3,3,2,0.1,0.2,0.8,0.1
U,4,4,2,0.2,0.8,0.1,0.4

```

```
#redundancies: Idx,fromD,toD,type(1=synch/2=std/3=erb),Rd
R,1,2,1,2,0.08
```

Quellcode 6.3: Beispiel eines Input-File

Wenn man die Elemente betrachtet, kann darin das System aus Abbildung 5.21a wiedererkannt werden: vier Komponenten in drei Erweiterungen, zwei Datenbestände, vier Nutzbeziehungen und genau eine *unmanaged* Datenredundanz mit dem Wert $\overline{Rd} = 0.08$. Wendet man nun darauf beispielsweise dreimal `extend()` und `analyse()` an, dann sieht die dabei produzierte *output.txt* wie folgt aus:

```
-------------------------------------------------------------
ARI:    2.0629957 (0.432 + 0.010354337 + 1.6206412) * q
k:      0.9889717
AV:     0.61129063 (std: 0.625)
TtM+:   4.13 days
DevC+: 1652.0 euros
Ext[3]: #FP = 185.0, #C = 4, #DB = 2, #Use = 4, #Red = 1
*************************************************************
Updated Database: d1 with E = 3133 in extension 0
Updated Component: c1 with FP = 21 in extension 0
New Use: c1 -> d2 (0.06,0.1,0.19,0.34)
Updated Database: d2 with E = 5542 in extension 2
-------------------------------------------------------------
ARI:    2.203925 (0.51012003 + 0.066525534 + 1.6272794) * q
k:      0.9882897
AV:     0.6104479 (std: 0.625)
TtM+:   4.41 days
DevC+: 1764.0 euros
Ext[4]: #FP = 186.0, #C = 4, #DB = 2, #Use = 5, #Red = 1
*************************************************************
New Database: d3 with E = 34601 in extension 5
Updated Component: c2 with FP = 112 in extension 1
New Use: c2 -> d2 (0.28,0.44,0.23,0.76)
Updated Database: d3 with E = 35749 in extension 5
New Red: d2 -> d3 with Rd = 0.06 (STD)
Updated Component: c1 with FP = 22 in extension 0
New Use: c1 -> d3 (0.47,0.68,0.22,0.15)
Updated Component: c4 with FP = 41 in extension 3
New Database: d4 with E = 34763 in extension 5
-------------------------------------------------------------
ARI:    3.7398558 (2.03525 + 0.96700656 + 0.73759943) * q
k:      0.9806965
AV:     0.60110354 (std: 0.625)
TtM+:   7.48 days
DevC+: 2992.0 euros
Ext[5]: #FP = 190.0, #C = 4, #DB = 4, #Use = 7, #Red = 2
*************************************************************
Updated Database: d1 with E = 3214 in extension 0
```

```
Updated Database: d2 with E = 5543 in extension 2
-------------------------------------------------------------
ARI:    3.7399707 (2.03525 + 0.96700656 + 0.73771423) * q
k:      0.9806959
AV:     0.60110277 (std: 0.625)
TtM+:   7.48 days
DevC+: 2992.0 euros
Ext[6]: #FP = 190.0, #C = 4, #DB = 4, #Use = 7, #Red = 2
*************************************************************
```

Quellcode 6.4: Das Output-File *output.txt*

Wie man erkennen kann, wurde nach jeder der drei Erweiterungen das System analysiert und die Metriken berechnet. Diese werden dann durch Werte wie `ARI`, `k`, `TtM+` und andere als Statistik ausgegeben. All die Werte, welche diese Arbeit theoretisch eingeführt hat, werden hierbei für jeden Systemzustand berechnet. Dies erlaubt es, den Wertgewinn oder -verfall nachzuvollziehen. Die Werte wurden für kleinere Systeme per Hand nachgeprüft und als richtig eingestuft, um eventuelle Fehler bei großen Systemen auszuschließen, bei denen man die Metriken nicht mehr sinnvoll per Hand ausrechnen kann. Das zugehörige und wesentlich kleinere *output_table.csv* sieht (gerundet und gekürzt) dann folgendermaßen aus:

```
FP,eFP,ARI,y_a,y_c,y_p,k,TtM,DevC,AV_new,AV_std
185.0,187.06299,2.062996,0.432,0.010354,1.6206412,0.988972,[...]
186.0,188.20392,2.203925,0.510,0.066526,1.6272794,0.988290,[...]
190.0,193.73985,3.739856,2.035,0.967007,0.7375994,0.980697,[...]
190.0,193.73997,3.739970,2.035,0.967007,0.7377142,0.980696,[...]
```

Quellcode 6.5: Das Output-File *output_table.csv*

Zuletzt wird das durch die Erweiterungen erstellte System noch herausgeschrieben und als fertige Datei *output_system.txt* produziert, welche das gleiche Format wie das Input-File besitzt:

```
#components: Idx,Ext,FP
C,1,0,22
C,2,1,112
C,3,2,15
C,4,3,41

#databases: Idx,Ext,E,*sigma,*sigma'
D,1,0,3214,0.0,0.0
D,2,2,5543,0.14,0.14
D,3,5,35749,0.0,0.0
D,4,5,34763,0.0,0.0

#uses: Idx,fromC,toD,phi_C,phi_R,phi_U,phi_D
U,1,1,1,0.1,0.3,0.3,0.7
U,2,2,1,0.0,0.0,0.4,0.2
U,3,3,2,0.1,0.2,0.8,0.1
U,4,4,2,0.2,0.8,0.1,0.4
U,5,1,2,0.06,0.1,0.19,0.34
U,6,2,2,0.28,0.44,0.23,0.76
```

```
U,7,1,3,0.47,0.68,0.22,0.15

#redundancies: Idx,fromD,toD,type(1=synch/2=std/3=erb),Rd
R,1,2,1,2,0.08
R,2,2,3,2,0.06
```

Quellcode 6.6: Das Output-File *output_system.txt*

Hier kann man erkennen, dass in fast jeder Kategorie ein oder mehrere Elemente geändert und deren Werte neu belegt wurden. Es sei noch angemerkt, dass ein interner *Seed* die Zufälligkeit steuert, sodass Experimente mit dem Tool mit den gleichen Parametern auch das gleiche „Endsystem“ erzeugen, um Wiederholbarkeit zu gewährleisten. Ebenso ist zu beachten, dass die Metrik-Werte q_a und q_c jeweils mit 0.5 belegt wurden (q_p wurde auf 1.0 belassen), um das Ergebnis näher an realistischere Werte heranzubringen. Warum dies getan wurde, wird im folgenden Abschnitt genauer erläutert.

6.3 Auswertung der Simulation

Hier wird erläutert, welche Erkenntnisse aus der Benutzung mit dem Tool gezogen wurden und inwieweit die beiden folgenden Fragen zu beantworten sind:

1. Verhält sich das Programm erwartungsgemäß, und werden die Metriken sinnvoll umgesetzt? (Effizienz des Programms)

2. Ist die zugrunde liegende Theorie des Graphen und der Metriken sinnvoll umgesetzt? (Effizienz der Theorie)

Diese Fragen werden in den beiden folgenden Abschnitten näher betrachtet. Bei der Auswertung der Fragen wurde als Input-File folgendes System benutzt:

```
#components: Idx,Ext,FP,eFP
C,1,0,20,0
C,2,1,110,0
C,3,2,15,0
C,4,3,40,0
C,5,4,70,0
C,6,4,60,0

#databases: Idx,Ext,E,*sigma,*sigma'
D,1,0,3000,0,0
D,2,2,5500,0,0
D,3,4,5000,0,0

#uses: Idx,fromC,toD,phi_C,phi_R,phi_U,phi_D
U,1,1,1,0.1,0.3,0.3,0.7
U,2,2,1,0.0,0.0,0.4,0.2
U,3,3,2,0.1,0.2,0.8,0.1
U,4,4,2,0.2,0.8,0.1,0.4
U,5,5,2,0.5,0.5,0.3,0.5
U,6,6,3,0.3,0.7,0.4,0.2
```

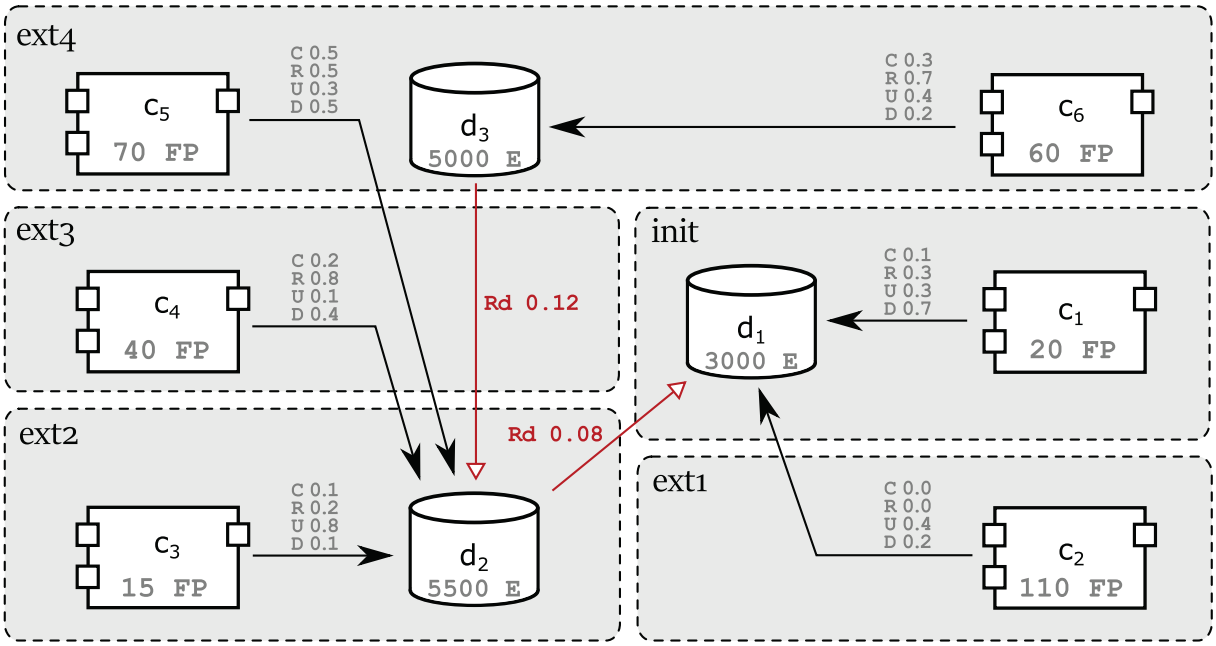


Abbildung 6.5: Die Ausgangssituation: das System als Input

```
21
22 #redundancies: Idx,fromD,toD,type(1=synch/2=std/3=erb),Rd
23 R,1,2,1,2,0.08
24 R,2,3,2,2,0.12
```

Quellcode 6.7: Das Input-File *input.txt* für das große Szenario

Dieses System stellt eine erweiterte Fassung des aus Abbildung 5.21a bekannten Systems dar und ist in vollem Umfang in Abbildung 6.5 aufgezeigt. Der Grund, warum nicht das gleiche System genutzt wird, ist der, dass Tests ergeben haben, dass die Messungen plausibler werden, wenn man die Simulation mit mehr Elementen startet. Das kann man durchaus damit begründen, dass Faktoren der Metriken, die vom Gesamtsystem abhängen, dann bereits besser hineinwirken, wie zum Beispiel der Median aller *Entities* oder die Summe aller *FP*'s. Aus diesem Grund wurde das System um einige Elemente erweitert, um möglichst ausgeglichene Analysen zu bekommen, besonders wenn man ohne vorher empirisch ermittelter Werte (*FP*, *E*, *Rd*, und anderes) für die wenigen Start-Elemente beginnt, wie in dieser Arbeit geschehen.

6.3.1 Bezug zur Simulation

Das Testszenario wurde dem weiter oben beschriebenen Workflow unterzogen. Insgesamt wurden 1000 Erweiterungen hinzugefügt und ausgewertet (wofür das Programm ungefähr 20 Minuten Rechenzeit benötigte). Die Parameter (aufgezeigt im Anhang unter A.2) wurden dabei so belegt, dass sie realistischen Werten nahe kommen. Das System, was dadurch entstand, ist zu groß, als dass es hier auf eine Seite sinnvoll abgebildet werden könnte. Um den Unterschied zu verdeutlichen, sind hier einmal beide Situationen als Ergebnis des `analyse()` repräsentiert:

```
1 ------------------------------------------------------------
2 ARI:    4.5803804 (1.608 + 0.034572285 + 2.9378083) * q
3 k:      0.9856675
```

```
AV:     0.6072128 (std: 0.625)
TtM+:   9.16 days
DevC+: 3664.0 euros
Ext[4]: #FP = 315.0, #C = 6, #DB = 3, #Use = 6, #Red = 2
*********************************************************

[... +1000 Erweiterungen]

_________________________________________________________
ARI:    1646.3567 (723.0082 + 410.60257 + 512.746) * q
k:      0.92344433
AV:     0.5329684 (std: 0.625)
TtM+:   3292.71 days
DevC+: 1317084.0 euros
Ext[1004]: #FP = 19859.0, #C = 352, #DB = 159,
           #Use = 3852, #Red = 335
*********************************************************
```

Quellcode 6.8: Das Output-File *output.txt* in stark gekürzter Fassung

Man kann erkennen, dass durch die Zufallsparameter das System mit 352 Komponenten, 159 Datenbeständen, 3852 Nutzbeziehungen und 335 Datenredundanzen fertiggestellt wurde. Das ergibt im Schnitt zwei *unmanaged* Datenredundanzen pro Datenbestand und ungefähr 11 ausgehende Nutz-Beziehungen pro Komponente. Wem das zu viel erscheint, kann die Zufallsparameter jederzeit abändern. Das Ergebnis ergab einen Verfall von ungefähr 7.6%, und damit einen Mehraufwand von 3300 Personentagen mit geschätzten Kosten von 1.3 Millionen Euro, wobei sich die 3300 Personentage bei einer 50-Mann-Firma vermutlich auf 65 „reale“ Tage an Mehraufwand pro Person belaufen. Am Schluss der Simulation konnte man durch das Output-File *output_system.txt* (hier nicht abgebildet) erkennen, dass beispielsweise die Komponente c_2 auf 316 *FP*, c_3 auf 51 *FP* und der Datenbestand d_1 auf 9149 *E* angewachsen ist. Wie genau die einzelnen Elemente erweitert werden, wird durch die Parameterbelegung festgelegt. Als Auswertung im Bezug auf die Simulation ist es wichtig, festzustellen, welches System entsteht. In Tests konnte ein extremer Unterschied bei der Veränderung einzelner Parameter festgestellt werden. Besonders hing die Gestaltung des Endsystems davon ab, inwiefern die einzelnen Elemente eine Erweiterung erfahren haben. In ersten Versuchen wurden stets neue Komponenten hinzugefügt, welche mehr oder minder den bereits existierenden Elementen im Ausgangs-System glichen. Dieser Ansatz wurde später verworfen, und es wurde das realistischere Verhalten gewählt, dass bereits existierende Elemente eher erweitert werden, als dass neue Elemente hinzugefügt werden. Das Verhältnis bei der Verteilung zwischen *Erweiterung* und *Neu* wurde dann auf 9 : 1 gesetzt. Das führte dazu, dass meistens die vorhandenen Komponenten einer stetigen (geringen) prozentualen Erweiterung der inhaltlichen Größe unterzogen wurden. Der Wert dafür lag zwischen 0.5% bis 2.0% der ursprünglichen Größe. Dies wurde mit dem realistischen Szenario begründet, dass eine Komponente in großen Systemen nicht immer komplett geändert wird, sondern eher nur kleine Teile davon ausgetauscht, beziehungsweise erweitert werden, um neue Funktionen der Komponente umzusetzen. Gleiches wurde für die Datenbestände angenommen.

Das Schwierige an der Arbeit mit zufällig erweitertem System ist, den Zufall so zu definieren, dass es einem natürlichen Prozess in der Softwareevolution gleicht. Die Hauptfrage war dabei immer: „Welche Parameter sind dafür notwendig und wie müssen diese belegt werden?“. Ein einfaches Beispiel soll dies kurz verdeutlichen. Angenommen, es werden stetig neue Komponen-

ten hinzugefügt. Inwiefern sollen dann auch gleichzeitig neue Nutzbeziehungen entstehen? Die Art und Weise, wie man die neuen Nutzbeziehungen in das System einbaut, ist entscheidend in Belangen der Theorie, da einige Metriken sehr sensibel auf viele neue Nutzungen reagieren, wie zum Beispiel das Nutzungsverhältnis $N(d_j)$ eines Datenbestandes d_j, welches dadurch beeinflusst wird, wie viele Nutzbeziehungen von Komponenten an dem Datenbestand ankommen. Die Problematik zu den Metriken soll im folgenden Abschnitt erläutert werden. Dies sollte nur auf das Problem hinweisen, dass ein deterministischer Algorithmus nur schwierig aus real existierenden Entscheidungen abzuleiten ist. Alleine die Art, wie eine Entscheidung in der System-Evolution in einem Parameter für die Zufallserweiterung umgesetzt wird, ist kritisch für den Erfolg. Hinzu kommt anschließend noch der zweite Punkt: eine sinnvolle Belegung für den Parameter zu finden.

Es war schwierig, eine Belegung aller Parameter zu finden, die das System in realistischen Maße wachsen lässt. Es ist nicht sinnvoll, die Auswertung der Theorie an einem System zu vollführen, welches sich zu sehr von „normalen“ Systemen unterscheidet. Aber gerade die Entscheidung, welches ein realistisches System sei, war schwierig, da die vorherige empirische Ermittlung fehlte, weswegen die Systeme auf Erfahrungen und Vorüberlegungen beruhen. Mit den am Ende festgelegten (in als anfänglicher Standard im Programm eingearbeiteten) Parameter ließen sich sinnvolle Prozesse der Evolution nachvollziehen. Sollte sich einmal eine andere Vorgehensweise der Evolution etablieren, dann ist es dennoch ein Leichtes, die Parameter der Zufallssteuerung dahingehend anzupassen. Die Parameter, so wie sie derzeit als Standard vorgegeben sind, eignen sich nach eigenen Testuntersuchungen gut, um ein realistisches Evolutionsszenario darzustellen. Jedoch sollte immer darauf verwiesen werden, dass das Programm mit seiner zufälligen Erweiterung des Graphen die Darstellung des Nachweises bildet, was ausschlaggebend für die Messbarkeit ist, die spätere Auswertung der Metriken durch das Programm. Man sollte sich also nicht wundern, wenn eine starke Veränderung an den Zufallsparametern eine starke Veränderung an dem Ergebnis der Metriken hervorruft.

6.3.2 Bezug zur Theorie

Die Auswertung des oben beschriebenen Szenarios erwies sich als schwierig. Wie erwähnt, ist die Messbarkeit stark abhängig von der Darstellung, und diese Darstellung wird von diversen deterministischen Zufallsparametern geführt. Deshalb wird hier, nachdem dazu im vorherigen Abschnitt das Problem bereits erläutert wurde, davon ausgegangen, dass das entstehende System nach den Standard-Parametern geeignet ist, die Messbarkeit durchzuführen, das heißt, die Metriken anzuwenden.

Als Ausgangspunkt diente das System aus Abbildung 6.5. Dies ist der „Sockel“, auf dem der gesamte Evolutionsprozess stattfindet. Es wurden 750 Erweiterungen ausgeführt, sodass das System mehr oder weniger kleine Änderungen erfuhr. Nach jedem Schritt, das heißt nach jeder Erweiterung $\mathbb{E}^+$, wurde ein `analyse()` ausgeführt, sodass alle Werte der Analyse (*ARI*, *k*, *AV*, *TtM* und *DevC*) nach jeder Änderung erfasst wurden, damit das System jederzeit vergleichbar bleibt. Neu ist die Erkenntnis über das Verhalten des Systems bei vielen Erweiterungen. Wie sich die Erhöhung einer *unmanaged* Datenredundanz auswirkt, wurde bereits in Kapitel 5.4.2 „Die Wert-Formel“ aufgezeigt. Im Gegensatz zu der lokalen Änderung an einer Redundanz, wo sich die Metriken wie in den Formeln beschrieben verändern, war es unbekannt, wie sich die Metriken verhalten, wenn das System selbst sehr vielen Erweiterungen mit allen Elementen unterzogen wird. Wie schon im vorherigen Abschnitt beschrieben, hängt dies sehr stark mit der Art und Weise zusammen, wie man das System tatsächlich erweitert. Zum Beispiel hat sich herausgestellt, dass die *Adaptive Maintenance* sehr stark anwächst, sobald viele Komponen-

ten hinzugefügt werden. Gleiches gilt für einen starken Zuwachs von Datenbeständen, was die *Preventive Maintenance* stark erhöht.

Nun galt es, das System zu werten und eine Aussage über die Stichhaltigkeit der Theorie zu treffen. Dazu werden die drei folgenden Punkte analysiert:

1. die Veränderung der einzelnen Metriken zu *Adaptive*, *Corrective* und *Preventive Maintenance*,
2. die Veränderung des *Systemwertes k*, des *Agility Values AV* sowie der beiden Kostenfaktoren *TtM* und *DevC*, und
3. die Bestätigung der Theorie in der Gesamtbetrachtung.

Auf alle drei Punkte soll näher eingegangen werden, zusammen mit der Beschreibung der Annahmen, Folgerungen und Ergebnisse. Wichtig ist noch zu erwähnen, dass dieses Kapitel die Funktionalität der Theorie mittels eines Programms ermittelt, nicht jedoch die Theorie selbst nachweisen soll, da dies nicht die Intention des Programms war. Dafür spielen zu viele Faktoren des Programms mit hinein, um eine solche Aussage treffen zu können.

Die Metriken Die drei Metriken werden hier getrennt voneinander betrachtet, da sie schließlich auch voneinander getrennt ausgerechnet werden. Dies soll die Unterschiedlichkeit der einzelnen Bereiche wiedergeben. Doch zuvor sei hier nochmals auf die drei Parameter hingewiesen, welche explizit für diesen Test angepasst wurden:

$$q_a = 0.5$$
$$q_c = 0.5$$
$$q_p = 1$$

Dies entstand nicht durch empirische Ermittlungen, sondern durch eine Normalisierung der Kurven der einzelnen Metriken. Dies bedeutet, obwohl keine echten Daten von Systemen vorlagen, wurden die Kurven in ihrer Höhe mit dem Unternehmensfaktor angepasst, sodass keine der drei Kurven in eine extreme Richtung abweicht. Zudem wurde darauf geachtet, dass am Ende ein sinnvoller Wert für den ARI entsteht. Dafür wurde der Unternehmensfaktor eingeführt, um spezifische Abweichungen der Resultate konstant in ihrer Höhe zu ändern und damit all jene andere Faktoren auszugleichen, welche keine direkte Beziehung durch die Metrik haben. Dabei kann bereits ein anderes Vorgehen der Evolution eine Änderung des Unternehmensfaktors nach sich ziehen, allein dadurch, dass Komponenten anders „gezählt“ oder hinzugefügt werden. Eine Änderung des Unternehmensfaktors ist immer einer Änderung an der Metrik vorzuziehen, da die Metrik logisch konsistent bleiben muss. Allen Metriken ist ebenso gemein, dass Erkenntnisse aus der Simulation diverse Auswirkungen auf die Metriken selbst hatten. So wurden einige Faktoren abgeschwächt, andere umgestellt oder in ihrem Inhalt verändert, je nachdem, wie das Gesamtsystem besser die Wirkung der Metriken widerspiegelte.

Die *Adaptive Maintenance* hatte sich als sehr stark erwiesen. Sobald viele Komponenten im System entstanden, verlief die Kurve für y_a in extreme Höhen (teilweise um das Zehnfache der anderen Kurven erhöht). Das liegt hauptsächlich an den FP, welche direkt hinein berechnet wurden, sodass eine Komponente mit 40 FP immer einen relativ hohen ARI-Wert erzeugte, was sich auf Dauer stark summierte. Dem wurde mit zwei Entscheidungen begegnet:

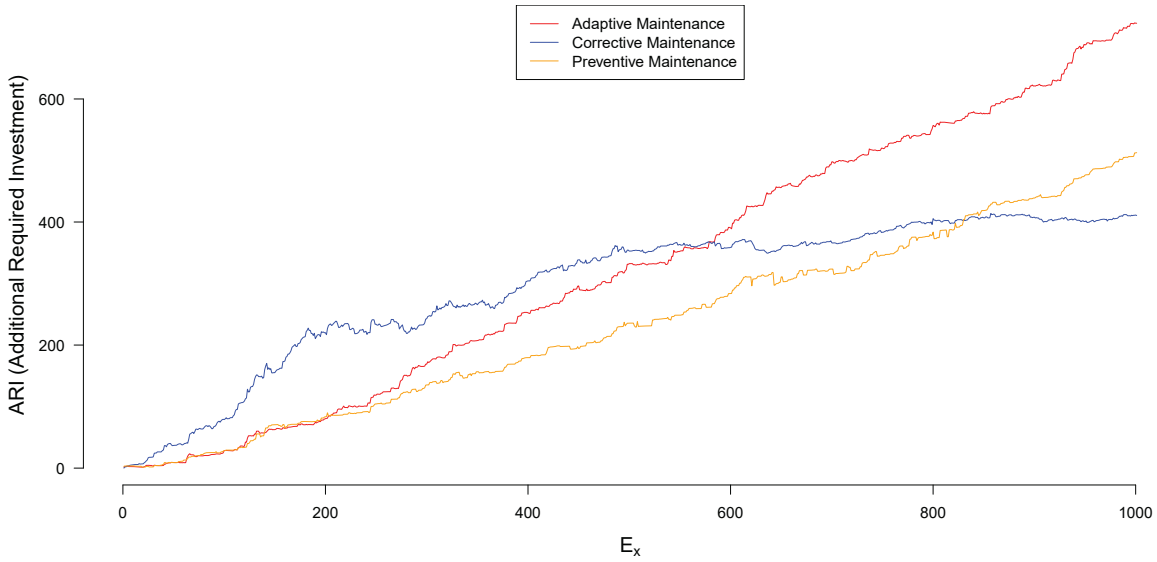
1. Die inhaltliche Größe FP wurde in $C(c_i)$ geändert (in der finalen Metrik zu sehen), was zu einer realistischeren Verteilung der „Erweiterungen“ an einer einzelnen Komponente sorgte.

2. Der Algorithmus zu der zufälligen Evolution des Systems wurde dahingehend angepasst, kleinere und realistischere Erweiterungen an Komponenten hinzuzufügen, sodass nicht mehr nur große und komplett neue Komponenten hinzugefügt werden.

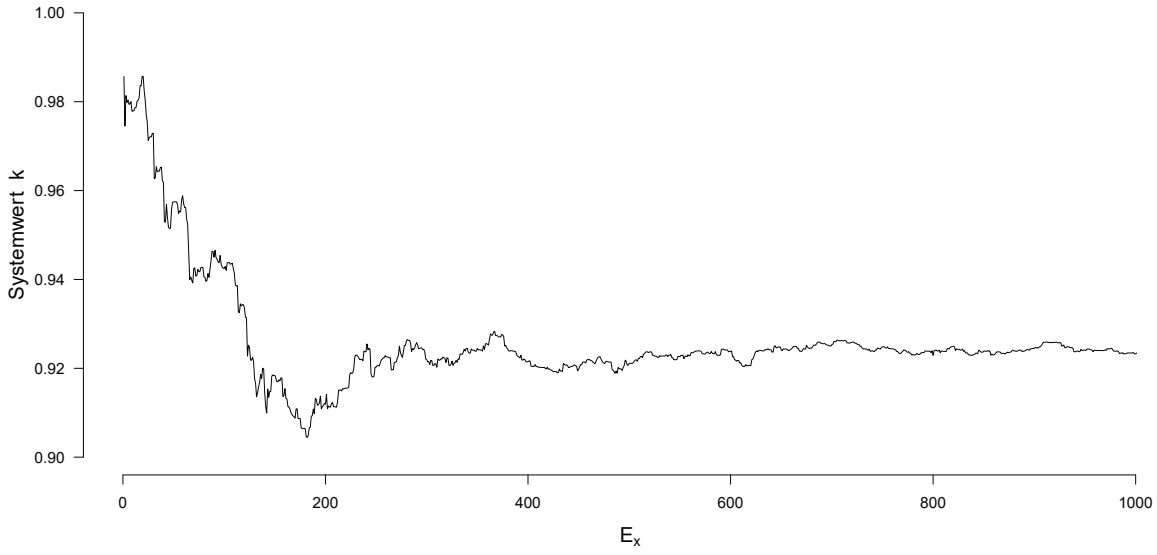
Die neue Kurve wurde mit $q_a = 0.5$ belegt, sodass sie im Verhältnis zu den anderen beiden Metriken steht. Das Resultat ist in Abbildung 6.6a zu sehen, in Form einer roten Kurve. Dass diese Kurve einen leicht exponentiellen Anstieg besitzt, liegt daran, dass Erweiterungen im System durch die Zufallsvorgabe in Prozent gegeben ist, sodass die prozentuale Erweiterung immer größer wird, je größer die eigentliche Komponente an sich bereits ist. Nur bei komplett neuen Erweiterungen wird ein absoluter Wert genutzt. Damit ist die Kurve (und damit der Anstieg) der *Adaptive Maintenance* in ihrer Natur eigentlich linear. Gleiches wird auch für die anderen beiden Kurven gelten, da das System immer linear (zufällig) erweitert wird. Schlussendlich spiegelt die Kurve der *Adaptive Maintenance* die Intention der Metrik einigermaßen sinnvoll wider.

Die *Corrective Maintenance* ist eine sehr spezielle Metrik, welche für viel Veränderung in der Metrik selbst sorgte. Oft wurde diese Metrik umgearbeitet (auch weil sie anfangs wirklich sehr kleine Werte produziert), was aber immer wieder zu der in den oberen Kapiteln beschriebenen Metrik zurück führte. Einerseits ist diese finale Metrik sehr anfällig für Veränderungen im System, wie man ebenfalls in Abbildung 6.6a erkennen kann, wo sich der lokale Anstieg der blauen Kurve stark ändert und auch direkte Auswirkung auf den Systemwert besitzt (wie in Abbildung 6.6b zu sehen). Das kann hauptsächlich dadurch begründet werden, dass zum einen die kompletten *FP* des Gesamtsystems einbezogen werden, zum anderen die Funktion $r(\lambda)$ sich nach und nach stärker auswirkt. Das führt dazu, dass sich Veränderungen im Verlauf ergeben, sobald eine Komponente stärker verändert wird oder eine neue Komponente hinzugefügt wird. Auch hierbei ist zu vermerken, dass die Kurve eigentlich linear ansteigt, jedoch durch die zufällige Evolution gegen Ende einen abgeschwächten Anstieg besitzt, da die später hinzugefügten Elemente des Systems nicht mehr ganz so drastisch im System „verwoben“ sind, wie die wenigen Anfangselemente, weswegen sich die Fehlerdichte verringert. Auch diese Metrik wurde mit $q_c = 0.5$ gedämpft, da die Auswirkungen sonst zu stark gewesen wären. Ansonsten muss q_c dazu benutzt werden, die Fehlerbehebungsmaßnahmen im Unternehmen auszudrücken, zum Beispiel, wie gut der Code dokumentiert ist und wie erfahren die Entwickler mit dem Code sind. Insgesamt ist die Auswirkung im großen zeitlichen Verlauf zufriedenstellend, da mit den gewählten Parametern ein sinnvoller Verlauf der Metrik entsteht.

Die *Preventive Maintenance* ist ebenfalls anfällig für Veränderungen im System, was hauptsächlich mit dem $\tilde{E}$ begründet werden kann, zumindest am Anfang der Evolution, da dann noch der Median häufiger wechseln kann. Insgesamt verläuft die Kurve (Abbildung 6.6a, gelbe Kurve) aber stabil, da kein reiner Bezug zum Gesamtsystem ($FP(\mathbb{E})$) hergestellt wird, welcher sich konstant erhöht. Trotz dessen, dass die *unmanaged* Datenredundanz $\overline{Rd}$ exponentiell in die Metrik eingeht, ist die Kurve aus Sicht der Evolution wieder linear, da die zufälligen Erweiterungen linear bestimmt werden. Die leichte Erhöhung des Anstiegs entsteht wieder aus der kleinen prozentualen Erweiterung der bestehenden Komponenten im System. Bei dieser Kurve kann man sich durchaus darüber uneinig sein, ob es realistisch ist, dass die *Corrective Maintenance* fast vollständig „unterhalb“ der anderen beiden Metriken liegt, was jedoch kein Anlass ist, dass der Verlauf der Kurve falsch sein könnte. Stellt sich heraus, dass die *Corrective Maintenance* oberhalb den anderen Kurven verlaufen müsste, kann man einfach $q_p = 2$ wählen, um dies realistisch darzustellen. Allerdings ist zu beachten, dass q_p auch stark davon abhängt, wie man als Unternehmen genau eine Synchronizität herstellt und wie viel Probleme dies bereitet. An sich ist die Metrik aber geeignet, da besonders am Anfang der *ARI*-Wert y_p besonders klein ist (zu sehen in Abbildung 6.7a), da zu diesem Zeitpunkt der Aufwand, die „wenigen“ Redundanzen zu



(a) Der zeitliche Verlauf der Wert-Metriken bis Erweiterung $\mathbb{E}^{+}_{1004}$



(b) Der zeitliche Verlauf des Systemwertes bis Erweiterung $\mathbb{E}^{+}_{1004}$

Abbildung 6.6: Die Auswirkungen der *unmanaged* Datenredundanz im System (1)

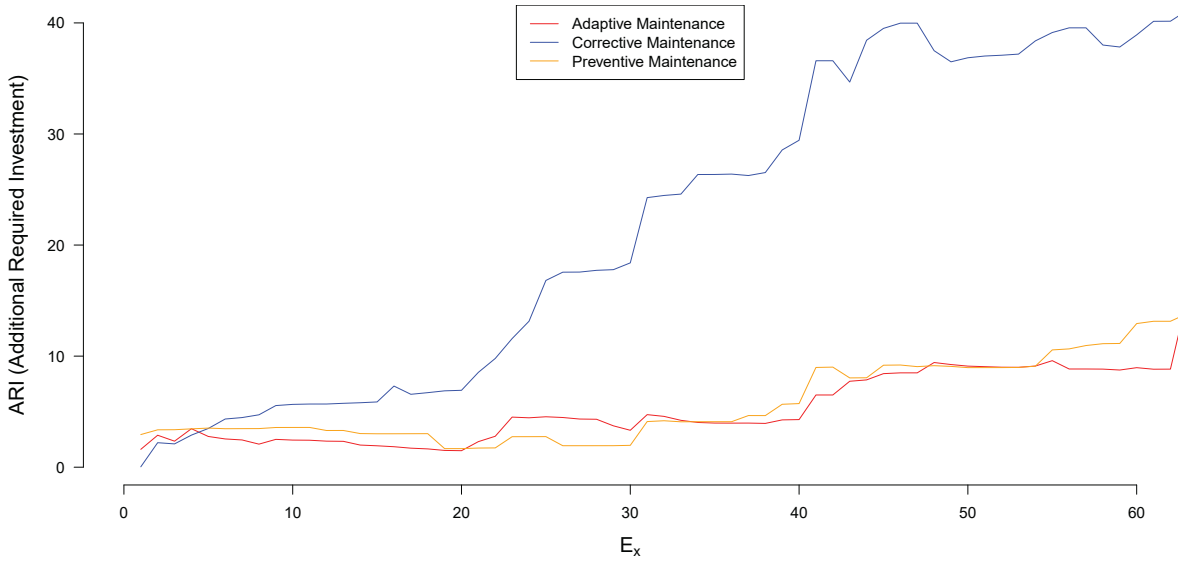
synchronisieren, meist nicht sehr hoch ist.

Aus Übersichtszwecken sind in Abbildung 6.7 noch zwei weitere Verläufe aufgezeigt. In Abbildung 6.7a kann man den Anfang der Simulation erkennen, bis hin zu der 60. Erweiterung. Währen in dieser Zeit die *Adaptive* und *Preventive Maintenance* sich noch klein halten, steigt die *Corrective Maintenance* schon stärker an. Dies kann man für mehr oder weniger realistisch halten, gerechtfertigt wäre es jedoch dadurch, dass dieses Verhalten das reale Verhalten im System darstellen *kann*. Während Anfangs nur wenige *unmanaged* Datenredundanzen vorhanden sind, häufen sich einerseits die dadurch verursachten Fehler, andererseits gäbe es bezüglich der Entfernung dieser wenigen Redundanzen noch nicht viel zu tun und auch die Erweiterung des Systems fällt noch leicht. Im späteren Verlauf des Systems, wenn die Fehler nur noch den kleinsten *ARI* einbringen, sind *Adaptive* und *Preventive Maintenance* stark angestiegen, da kaum noch ungehindert neue Erweiterungen getätigt werden können und das Entfernen der *unmanaged* Datenredundanz ein riesiger Aufwand ist, wohingegen die reine Fehlerkorrektur nur noch klein und praktikabler zu sein scheint (was auch meist der reale Grund dafür ist, dass Systeme als *Fossil System* enden, siehe dazu Kapitel 2.3 „Fehler durch Architektur und Management“). In Abbildung 6.7b sieht man, dass sich der Unterschied zwischen den normalen *FP* und den *eFP* immer weiter vergrößert. Und obwohl sich der Systemwert k in der zweiten Hälfte nicht mehr stark abändert (d.h. der Systemwert stagniert im Unendlichen), steigen die *eFP* kontinuierlich an, was damit auch erhöhte Kosten und Zeit in Form von *TtM* und *DevC* verursacht. Das ist dadurch begründet, dass der Systemwert k ein relativer Wert ist, wohingegen der *eFP*-Wert des Systems immer absolut zählt, unabhängig von der eigentlichen *FP*-Größe des Systems. Es ist zu beachten, dass das System mit fortschreitender Evolution hier sehr viele *FP* besitzt, was nicht auf alle Systeme zutrifft und teilweise unrealistisch sein kann. Dann empfiehlt es sich, die Parameter der zufälligen Erweiterungen dahingehend anzupassen. Das Verhältnis zwischen *FP* und *eFP* wird sich jedoch nur minimal ändern.

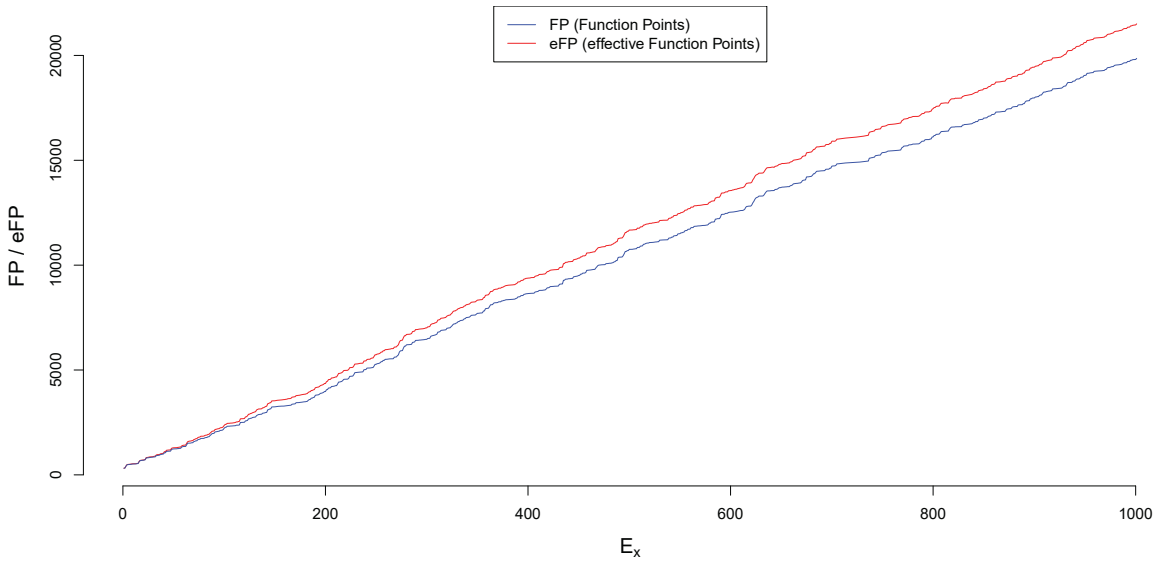
Insgesamt kann man mit den Metriken zufrieden sein, da keine ungewöhnlichen oder unerklärliche Phänomene auftraten. Man muss sich durchaus bewusst sein, dass alle drei Metriken stark davon abhängen, welche Zufälligkeit der Evolution gewählt wird, das heißt, wie alle Elemente im Graphen tatsächlich durch den Zufall gehandhabt werden. Eine Änderung im Zufall ergibt meist eine komplett neue Situation zwischen den Metriken. Deshalb ist es notwendig, vor der Anpassung der Unternehmensfaktoren oder gar der Metriken selbst, eine geeignete Repräsentation der zufälligen Erweiterung im Sinne einer realistischen Evolution zu finden. Dem kommt zugute, dass die meisten Parameter für Zufälligkeit durch eine natürliche Verteilung [64] gewählt werden können, was eine einigermaßen natürliche Evolution sicherstellen kann. Sobald man die Parameter so belegt hat, wie es den eigenen Vorstellungen entspricht, kann man die Unternehmensfaktoren anpassen.

Der Systemwert Durch die drei Metriken wird der Systemwert k definiert (vgl. Kapitel 5.4.2 „Die Wert-Formel“). Dieser ist für die Simulation bereits berechnet worden und in Abbildung 6.6b dargestellt. Als „verarbeitete Summe“ aller drei Metriken agiert der Systemwert als Vergleichspunkt zwischen zwei Systemzuständen τ und $\tau + t$, ähnlich dem *Agility Value*. Da nach jedem `analyse()` auch eine Berechnung des Systemwertes k erfolgt, kann man durch eine Verbindung aller Werte die in Abbildung 6.6b dargestellte Kurve bilden. Man beachte hierbei, dass sich der Wert für k hauptsächlich im Bereich $0.9 < k < 1$ befindet. Zur besseren Darstellung wurde auf der Y-Achse auch nur dieser Bereich abgetragen.

Man kann erkennen, dass die Eigenheiten der einzelnen Metriken sich durchaus direkt auf den Systemwert auswirken. Dadurch entsteht die Frage: Warum kann sich der Systemwert wieder



(a) Der anfängliche Verlauf des *ARI* von $\mathbb{E}_0^+$ bis $\mathbb{E}_{60}^+$



(b) Der zeitliche Verlauf der inhaltlichen Größe (*FP/eFP*) bis Erweiterung $\mathbb{E}_{804}^+$

Abbildung 6.7: Die Auswirkungen der *unmanaged* Datenredundanz im System (2)

erhöhen? Diese Frage ist dadurch geklärt, dass der Systemwert (wie auch andere Vergleichswerte wie AV) immer relativ zu sehen sind. Wie viel X bekommt man für so und so viel Y?

$$W = \frac{X}{Y}$$

Bekommt man mehr X für Y oder benötigt man weniger Y für die gleiche Menge an X, steigt das Verhältnis an, umgekehrt analog. Speziell für den Systemwert k gilt: wenn das System im Laufe der Zeit mehr *FP* generiert aber keine weiteren Fehler hinzukommen, steigt das Verhältnis dadurch an, dass der „Schaden vernachlässigbar“ wird. Das bedeutet, sobald die Kurve einen positiven Anstieg besitzt, sind neue Funktionalitäten und Erweiterungen hinzugekommen, während die Fehler gleich blieben oder negiert wurden. Allerdings ist es nie mehr möglich, $k = 1$ zu erreichen, da immer ein Rest an „Schaden“ im System verbleibt. Würde man dagegen das System fixieren und nur neue *unmanaged* Datenredundanz einbauen oder vorhandene erhöhen, würde der Systemwert wie erwartet stetig abfallen, da keine neue Funktionalität generiert wird. Da der Systemwert alle relevanten Metriken vereint, ist es ein guter Punkt, über den Verlauf der Evolution zu diskutieren. Ist die Kurve stetig am Abfallen, wird mehr Schaden generiert als für die neue Funktionalität nötig wäre. Steigt der Systemwert an, dann wird zumindest kein neuer relevanter „Schaden“ generiert. Dieser „Schaden“ bedeutet hierbei eine erhöhte Investition für „quasi null“ neue Funktionalität, nur als reine Reparaturkosten.

Tatsächlich ist es plausibel, dass der Systemwert „nur“ im oberen Zehntel des Wertes liegt, da man immer bedenken muss, dass hierbei nur die *unmanaged* Datenredundanz betrachtet wird. Viele andere Schadensquellen können einen zusätzlichen Beitrag leisten. Zudem mag ein Systemwert von $k = 0.992$ auf den ersten Blick nicht wirklich schadhaft erscheinen, aber bei genauerer Betrachtung kann sich das durchaus auf die Qualitätsattribute auswirken. Am ehesten kennt man diese Notation aus der *Hochverfügbarkeit* einer Software, bei der es tatsächlich einen großen (und kostenintensiven) Unterschied ergibt, ob das eigene System 99.9% oder doch 99.999% im Jahr verfügbar sein soll [25]. Genau so kann man den Systemwert k betrachten, der bis auf mehrere Nachkommastellen relevant werden kann, je nach Größe und Wichtigkeit des Systems.

Aber um bei realistischen Betrachtungsweisen zu bleiben, wird hier empfohlen, die Metriken nicht „auszureizen“. Es ist anzunehmen, dass das Nutzen von *unmanaged* Datenredundanzen von $\overline{Rd} > 0.3$ im System ein falsches Bild ergibt, da diese Werte zum einen unrealistisch sind, und zum anderen ihr Potenzial haben, zu stark in Metriken einzufließen. Welche sinnvollen Grenzen die Metriken haben, dass sich der Systemwert noch glaubhaft verhält, muss noch ausgetestet werden. In der Annahme steht derzeit, dass die Metriken gut funktionieren, wenn sich die Redundanzen im Bereich um ≈ 0.1 bewegen und die inhaltliche Größe von Komponenten und Datenbestände in der Evolution keine großen Differenzen aufweist.

Ansonsten ist durch diverse Faktoren in den einzelnen Metriken zu beobachten, dass sich der Systemwert k erst nach einigen Iterationen in eine stabile Lage begibt, in der Abweichungen seltener stark ausgeprägt sind, was durch die erhöhte Menge an Informationen über das System begründet ist. Der Systemwert verhält sich dadurch wie vorerst erwartet, variiert jedoch stark, wenn diverse Faktoren der Metriken anders ausgelegt werden sollten. Sobald eine geeignete Belegung aller Faktoren und Parameter gefunden wurde, dient der Systemwert k dazu, das System im Laufe der Zeit einschätzen zu können.

Die Theorie Die Auswertung des Programms in Bezug auf die Theorie ist recht kurz, da viele Teile dessen bereits in den vorherigen Abschnitten erklärt wurden. Hier soll nur noch einmal erfasst werden, inwieweit das Programm die Theorie stützen kann.

Viele kleinere Systeme, wie das in Abbildung 6.5 gezeigte System, kann man durchaus manuell ausrechnen. Die Formeln dazu werden vermutlich etwas länger, aber es ist möglich. Dahingehend sollte das Programm die Theorie nicht nachweisen, sondern beobachten, wie sich die Theorie bei sehr großen Systemen auswirkt. Und das ist mit dem erstellten Programm gelungen. Dabei konnten die folgenden Erkenntnisse zusammengefasst werden:

1. Die Theorie gilt vorzugsweise bei kleineren Systemen, wo sich die zufälligen Änderungen in Grenzen halten.

2. Bei großen Systemen unterscheiden sich die Metriken im Kurven-Verhalten nicht mehr allzu sehr voneinander.

3. Die *ARI*-Auswirkungen der Metriken bei großen Systemen können sich stark ändern, je nachdem, wie das System entsteht und welche Annahmen bei der Evolution zutreffen.

4. Der *Systemwert k* und der *Agility Value AV* verhalten sich identisch, da beide auf den gleichen Werten aufbauen. Erst wenn den *TtM* und *DevC* (teilweise) unabhängig des berechneten *ARI* sind, entstehen unterschiedliche Verhaltensmuster in den Kurven.

5. Der neue Gesamtaufwand steigt proportional mit dem „normalen“ Aufwand des Systems, wie in Abbildung 6.7b zu sehen. Dabei ist der „Unterschied“ in beiden Verläufen abhängig von den oben genannten Parametern und und der Evolution.

Insgesamt ist das Programm für die Auswertung größerer Systeme geeignet, wobei man sich immer bewusst sein muss, dass die bereits oft erwähnte Zufälligkeit einen erheblichen Einfluss ausübt. Tatsächlich würde man realistischere Werte erhalten, wenn man ein System abstrahiert nachbaut, was sehr aufwendig werden kann. Jedoch fließen bei dem „Selberbauen“ diverse Entscheidungen in das System ein, welche der Algorithmus für die Zufälligkeit nicht beachten kann, beispielsweise eine realistische Verteilung der Nutzbeziehungen. Ansonsten kann man aber sehr gut die Auswirkungen auf nicht mehr manuell handhabbare Systeme betrachten. Da das Programm mit seinen ungefähren 1300 *LOC* recht klein und methodisch getrennt aufgebaut ist, ist es dementsprechend anpassbar bei eventuellen Änderungen an den Metriken. Das Programm konnte die Theorie insgesamt dahingehend unterstützen, dass die Metriken durchaus auch auf eine lange Evolution und große Systeme angewendet werden können.

7 Schlussfolgerung und Abschluss

Nun, wo alle notwendigen Schritte der Theorie aufgezeigt und per Simulation nachgeprüft wurden, sollen in diesem letzten Kapitel offene Fragen sowie bereits vorhandene Kritik an der Theorie aufgezeigt werden, um diese Arbeit als Grundlage für weitere Forschung zu betrachten. Zusätzlich werden einige Punkte genannt, an denen direkte weiterführende Arbeiten denkbar sind, um das Konzept um die hier erbrachte Theorie entweder weiter zu stärken oder so abzuändern, dass sie der Realität noch mehr angenähert wird. Zum Schluss wird eine Zusammenfassung der Arbeit und der Theorie gegeben, zusammen mit einer Einschätzung des Autors zu der erstellten Arbeit.

Die Methodik und Vorgehensweise dieser Arbeit ist hierfür noch einmal kurz in Abbildung 7.1 aufgezeigt, als „Neben-Erkenntnisse“ und Vorgänger zu den Haupterkenntnissen und dem Ziel der Arbeit. Diese Methodiken, welche im Verlauf der Arbeit genutzt wurden, konzentrieren sich hauptsächlich auf die vier dargestellten Abschnitte. Angefangen mit dem *System als Graph* wurde aufgezeigt, wie man ein System als strukturierten und formalen Graphen darstellen kann, ebenso wie die Systeme auf die Menge der Komponenten, Datenbestände und Beziehungen eingegrenzt wurden. Die zweite Methodik waren die *Graph-Metriken*, welche dem Graphen inhaltliche Bedeutung verliehen hatten, indem sie die Beziehungen und Elemente im Graphen beschrieben. Es folgte die Messbarkeit auf der Darstellung mit dem Konzept der *eFP* als *Aufwands-Abschätzung* im System, welche erstmalig unabhängig vom realen System und nur auf Grundlage dessen Modells entstand. Um diese schlussendlich zu berechnen, kam das Konzept der *Wert-Metriken* zum Einsatz, welche genau diese *eFP* als *ARI* angeben, indem spezifische Faktoren aus der Darstellung genutzt wurden. Man beachte, dass dieser grafische Überblick kein Vorgangs-Diagramm darstellt, sondern die vier wichtigsten eingeführten Konzepte und Methodiken aufzeigt. Für einen schematischen Ablauf des Nachweises empfiehlt sich Kapitel 5.1 „Problematik und Methodik“ und nächstfolgende.

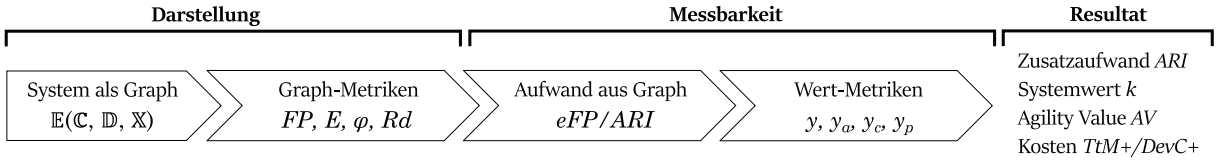


Abbildung 7.1: Die einzelnen eingeführten Methodiken für den Nachweis

7.1 Kritik und offene Fragen

Dem Autor ist bewusst, dass in dieser Arbeit nicht alle Aspekte perfekt abgearbeitet und Problemstellungen erfolgreich gelöst wurden. Wie bereits in Kapitel 5.6 „Fehleranalyse und Toleranzen“ beschrieben, gibt es einige Punkte, an denen eine gezogene Schlussfolgerung zumindest nicht als einzige Option möglich scheint. In diesem Abschnitt sollen kurz die vordergründigen Kritikpunkte und offenen Fragen bezüglich der in dieser Arbeit ausgearbeiteten Theorie aufgezeigt werden.

Wie lässt sich ein System nur in FP darstellen, wenn es auch Datenbanken benötigt? Dies ist eine Frage, die auf das Fundament des Nachweises abzielt. Obwohl die *FP* als geeignete Methode für den inhaltlichen Umfang einer Software gilt, weiß man aus Erfahrung in Projekten, dass der Aufwand aus mehr als nur dem reinen Inhalt besteht. Dazu sei angemerkt, dass die Berechnung von *FP* bereits Datenhaltung beinhaltet. Es ist schwierig, eine Größe eines Systems abzuschätzen, welche man gleichzeitig auf *TtM* und *DevC* abbilden, aber auch aus numerischen Metriken erfassen kann. Dafür eigenen sich die *FP* gut, wobei in dieser Arbeit einfachheitshalber davon ausgegangen wird, dass, wenn ein neuer Datenbestand entsteht, dies in dem Aufwand der zugehörigen Komponente eingerechnet ist. Immerhin gibt es bei der Entstehung des Datenbestandes auch die zugehörige *Delta-Ausnahme* δ in der *Adaptive Maintenance*, welche die schädliche *unmanaged* Datenredundanz vorerst ignoriert.

Die Definition von Komponente und Datenbestand ist sehr weitreichend. Wie passt dies zu generell gültigen Metriken für die Kosten? Besonders in der System-Simulation hatte sich herausgestellt, dass die Art und Weise, wie Komponenten und Datenbestände beschaffen sind und wie sie evolutionär weiterentwickelt werden, einen großen Einfluss auf die Auswirkungen der Metriken besitzt. So verhalten sich die Metriken bei kleinen und hauptsächlich erweiterten Elementen größtenteils anders als bei großen und eher neu erstellten Elementen. Dieser Umstand ist leider nicht zu vermeiden, da jedes Unternehmen und jedes System seine eigene Sichtweise und Evolution benötigt. Doch genau für diesen Fall sind die Metriken austauschbar gestaltet, sodass neue Kontexte auch anders eingearbeitet werden können. Sobald eine passende Parametrisierung für den aktuell anliegenden Kontext gefunden wurde, können die Metriken genutzt werden. Allerdings kann (noch) nicht beansprucht werden, dass die Metriken generell für alle möglichen Szenarien mit Parametrisierung gültig sind. Es ist durchaus vorstellbar, dass einige Systeme eine Definition von Elementen besitzt, welche zu sehr von den Voraussetzungen dieser Arbeit abweicht, beispielsweise das Abbilden von einzelnen Funktionen auf „Komponenten“ c, unabhängig davon, ob die *FP* dann noch berechnet werden könnten. Im Allgemeinen muss man daher das System, wie es in dieser Arbeit beschrieben wird, als abstraktes Modell betrachten, ohne den Zwang, es auf ein existierendes System-Modell übertragen zu können, denn das kann nicht garantiert werden und dafür wurde der System-Graph mit seinen Elementen auch nicht entwickelt.

Die Abbildung von FP auf Kosten in Form von TtM und DevC ist sehr tolerant. Es ist ein längerer Weg, der von den initialen *FP* einer Komponente zum Systemwert k führt. Auf diesem Weg werden viele Annahmen und Faktoren einbezogen, welche durchaus weitreichende Grenzen besitzen. Dabei ist es schwierig, diese Grenzen per Definition auf einen sinnvollen Grat einzugrenzen, sodass sich letzten Endes ein durchgängiger und unausweichlicher Pfad einer Beweiskette ergibt, besonders bei so theoretischen und abstrakten Themen wie Softwarearchitektur. Dass sich am Ende die *FP* einer Komponente in eine bestimmte Anzahl an „hartem Cash“ oder ein Programmieraufwand in Stunden abbilden lässt, ist das Resultat einer Vorgehensweise der Argumentation. Sollte darin eine der vielen möglichen Grenzen nicht gut genug betrachtet worden sein, ist anzunehmen, dass dies eine verfälschte Anzahl an Euro oder Stunden ergibt. Um das zu umgehen, sind viele empirische Studien und Nacharbeiten am Gesamtkonzept notwendig, welche nicht in dem Zeitrahmen dieser Arbeit möglich waren. Daher ist das Resultat der zusätzlich notwendigen *TtM* und *DevC* nicht ideal, jedoch in Hinblick auf einen kausalen Zusammenhang haltbar. Durch geeignete Veränderung der Grenzen aller Annahmen und Faktoren scheint es zumindest möglich, realistische Werte für die Kosten des abstrakten Systems durch

die Metriken zu erhalten, auch wenn der Zweck der Theorie nicht die exakte Vorberechnung der Projektkosten ist.

Ist die Einschränkung auf die eingeführten Elemente des Systems zu strikt für eine Kosteneinschätzung? Die Wahl, nur die vier Elemente (Komponente, Datenbestand, Nutzbeziehung und Datenredundanz) im Graphen zuzulassen, basiert auf dem Minimalprinzip. Diese vier Elemente sind theoretisch die notwendigen Mittel, um einen kausalen Zusammenhang zwischen Softwarearchitektur (oder eher der *unmanaged* Datenredundanz) und dem Wert eines Systems zu ziehen. Alles weitere hat das Potenzial, das Ergebnis in eine besondere Richtung abzuändern.Allerdings lässt sich nicht bestreiten, dass sich mit mehr Elementen oder mehr Systemeigenschaften eventuell genauere Resultate der Quantifizierung errechnen lassen. Darunter würde jedoch wiederum die Kausalität leiden, weil mehr Faktoren in jede einzelne Überlegung einbezogen werden müssten. Um einen durchschaubaren Nachweis zu liefern, wurde hierbei zugunsten der Nachvollziehbarkeit und der Kausalität des Nachweises entschieden und eventuelle Ungenauigkeiten bei der Quantifizierung in Kauf genommen. Wie eine ideale Vereinbarkeit aussehen könnte, könnte Thematik einer zukünftigen Arbeit sein.

Hilft der Wertverlust durch unmanaged Datenredundanz tatsächlich bei der Einschätzung der zukünftigen Systemkosten? Nicht ganz. Wie bereits geschildert, ist diese Arbeit primär kein Werkzeug, um die Kosten eines Projektes beim Entwickeln abzusehen, auch wenn die Quantifizierung Teil davon ist. Der Hauptaspekt liegt hierbei, wie bei vielen Beweisen und Nachweisen, auf der reinen Machbarkeit. Wie in Kapitel 1.2 „Motivation für den Nachweis von Werten der Architektur“ beschrieben, ist es der Sinn eines Beweises, etwas aufzuzeigen, um letztendlich Sicherheit über eine bestimmte Problemstellung zu erlangen. Später kann man darauf aufbauen und „Werkzeuge“ auf Grundlage des Nachweises bilden. Was hierbei gezeigt wurde, ist neben dem kausalen Zusammenhang auch eine Quantifizierung der *unmanaged* Datenredundanz, um das Bewusstsein über fehlerhafte Strukturen und Architekturen im System zu stärken. Dabei soll das Resultat des Systemwertverfalls nicht hauptsächlich dafür benutzt werden, Projekte abzuschätzen, sondern fehlerhafte Architektur während der Evolution zu verhindern, indem die Ausmaße der Schäden aufgezeigt werden. Allerdings wäre es durchaus sinnvoll, die Quantifizierung in zukünftigen Arbeiten dahingehend zu stärken, um diese zu einem belastbaren Faktor bei der Kalkulation werden zu lassen.

Der Wert des Systems wird ungeachtet des inneren Aufbaus der Elemente ermittelt. Das Problem, welches sich ergibt, wenn man versucht, ein reales System in den hier beschriebenen Graphen abzubilden, ist die Erkenntnis, dass dies meistens nicht wirklich funktionieren wird. Der Grund dafür ist relativ einfach: es werden zu viele Elemente eines realen Systems vereinheitlicht oder ungenutzt ausgelassen. Dieser Kritikpunkt ist berechtigt, weshalb immer darauf Wert gelegt wurde, zu betonen, dass dieses System auf Basis des Graphen ein abstraktes System darstellt, welches nicht einfach aus einem real existierenden System herleitbar ist oder aus welchem man einfach ein reales System bauen könnte. Alleine die Redundanz-Beziehungen sind sehr schwer in realen Systemen zu finden und dann derartig abzubilden. Aus diesem Problem des fehlenden realen Bezugs ergibt sich außerdem, dass bestimmte Situationen auf verschiedene Weise im Graphen gelöst werden könnten. Das Simulationsprogramm, welches für diese Arbeit erstellt wurde, kann wie in Abbildung 6.4 dargestellt werden, es gäbe jedoch vermutlich noch einige andere Variationen, die ebenso geeignet wären. Durch die Vereinheitlichung von Elementen des Systems geht auch deren Spezifika verloren. Man kann nicht verneinen, dass es keine

Rolle spielen würde, inwieweit eine Komponente in sich aufgebaut ist, wie gut ein Datenbestand strukturiert ist und welche Möglichkeiten er für die Synchronisation besitzt. Es wird nicht mehr unterschieden, ob eine Komponente eine Klasse darstellt, oder ein komplexes Subsystem. Dies kann theoretisch alles Auswirkungen auf eine Quantifizierung haben, ist jedoch nur von geringer Bedeutung bei dem Nachweis der Kausalität, dem Hauptaspekt dieser Arbeit. Wie schon des Öfteren geschrieben, bestünde bei zukünftigen Arbeiten die Möglichkeit, die Darstellung und die Messbarkeit des Nachweises zu verfeinern, indem der Graph oder die Metriken angepasst werden, um so die Quantifizierung exakter zu gestalten. Dabei können dann auch diverse innere Zustände der Knoten einfließen.

Inwieweit sind die Metriken vollständig? Die Metriken als ein Kernstück der Arbeit zielen primär darauf ab, den Schaden im System „in Worte zu fassen“. Bei der Erstellung wurde dabei darauf geachtet, möglichst konsistente Entscheidungen einzubeziehen. Die Metriken sind immer nur so vollständig, wie es

1. die Darstellung erlaubt, und

2. die Messbarkeit erfordert.

Erweitert man die Darstellung, kann das Spektrum der Metriken erweitert werden. Verfeinert man die Messbarkeit, präzisiert man dieses Spektrum wieder. Damit ergibt sich mit der Verbesserung der Theorie ein immer besserer Ansatz für eine Einschätzung des Systems durch die neuen Metriken. Es gibt nichts, was gegen eine Erweiterung der Redundanz-Beziehung Rd im Graphen spricht, womit später in den Wert-Metriken die erweiterte Fassung eventuell auch besser hinsichtlich des Gesamtaufwandes ARI ausgewertet werden kann. Allerdings ist es auch nicht falsch, die jetzige Version als „Basis-Variante“ zu nutzen, um die Quantifizierung anzugeben. Die jetzigen Metriken sind das, was der Graph als Darstellung hergibt und was in den Wert-Metriken eingearbeitet werden musste. Es sei denn, die Theorie wird widerlegt, was allerdings bei allen naturwissenschaftlichen Beweisen auftreten kann, bei den Einen mehr, bei den Anderen weniger wahrscheinlich. Damit sind die Metriken vermutlich niemals wirklich „vollständig“, sondern immer mehr oder weniger „geeignet“.

Inwieweit wird für getroffene Annahmen und Werte garantiert? Für die Werte und Verhaltensweisen kann nicht garantiert werden. Obwohl diese Arbeit einen kausalen Nachweis bildet, sind es doch sehr viele Annahmen und Einschätzungen, die einbezogen wurden, welche jeweils das Potenzial haben, den Ausgang zu verändern. Letztlich bleibt nur die Möglichkeit, die hier vorgestellten Erkenntnisse empirisch zu bestätigen oder zu widerlegen und die Theorie entsprechend zu verfeinern oder anzupassen.

Was ist, wenn ein Fehler in der Beweiskette auftaucht? Sollte ein Fehler im Nachweis entstehen, durch eine falsche Annahme, einen falschen Faktor in einer Metrik oder einer unrealistischen Darstellung des Systems als Graph, dann ist zwar die Sicherheit im Ergebnis nicht mehr gegeben, jedoch ist der Vorteil an naturwissenschaftlichen Beweisen, dass man diese anschließend entsprechend abändern kann, soweit möglich. Sollte ein Fehler in der Beweiskette dieser Arbeit enthalten sein, dann gibt es keinen Grund, alles zu entwerten. Die Vorgehensweise wurde explizit so gewählt, dass in jedem Punkt des Nachweises variable Parts vorhanden sind, sodass bei einer Änderung des Kontextes die weitere Vorgehensweise entsprechend angepasst werden kann, sei es

in der Darstellung oder in der Messbarkeit. Möglicherweise stimmen bei einem Fehler die errechneten Werte nicht mehr oder der kausale Zusammenhang ist gar nicht mehr gegeben, dann ist es dennoch sehr wahrscheinlich, dass man mit der hier gewählten Vorgehensweise über Graph- und Wert-Metriken das Ziel erreichen und einen Zusammenhang zwischen Architektur und Wert herstellen kann. Dann kann man diese Arbeit als Grundlage für die Ausbesserung nutzen.

7.2 Weiterführende Möglichkeiten

Nachdem die offenen Punkte der Arbeit erläutert wurden, stehen noch einige explizite Punkte im Raum, welche als Ansatz oder Ausgangspunkt für zukünftige Arbeiten dienen könnten. Diese weiterführenden Themen sollen hier noch einmal genannt werden.

Ermittlung von empirischen Werten für die Analyse der Theorie Die hier vorgelegte Theorie wurde aus Überlegungen erstellt, welche auf keiner empirischen Datenbasis beruhen, sondern auf langjährigen Erfahrungswerten in der Evolution von sehr großen Softwaresystemen beruhen. Eine reale empirisch ermittelte Datenbasis wäre ein wirklich großer Mehrwert für die hier vorgestellte Methode des Wert-Nachweises, da dadurch neue Erkenntnisse in die Darstellung und in die Messbarkeit einfließen können. Eine realistischere Einschätzung einzelner Faktoren kann eine Wert-Metrik verbessern und damit die Quantifizierung stärken. Eine wichtige Systemeigenschaft kann zusätzlich im Graphen notiert werden, um die Graph-Metriken zu verbessern, was die Resultate stützen würde. Üblicherweise kann man empirische Messungen bei naturwissenschaftlichen Beweisen voraussetzen. Dies war in dieser Arbeit leider nicht gegeben, da es nicht in den vorgegebenen Zeitrahmen machbar gewesen wäre, entsprechende Studien vorzubereiten. Stattdessen wurde entschieden, dass nur der theoretische Part angenommen wird, da hier der interessante Teil des Themas liegt und zudem der Mehrwert erreicht wird, als wenn man stattdessen nur die Studien erledigt und die Theorie dafür offen gelassen hätte. Würde man diese empirischen Studien nachholen, könnten genauere Erkenntnisse darüber erzielt werden, inwiefern verschiedene Konzepte in dieser Arbeit valide sind und welche Aussagekraft sie besitzen, wie zum Beispiel das Konzept der *Entities* mit der Kapselung von Informationen, die bloße Unterscheidung zwischen Komponenten und Datenbeständen oder die Sinnhaftigkeit der Nutzbeziehung zwischen beiden Elementen. Würde man diese und andere Konzepte mit statistischen Werten untermauern, dann würde alles einen wesentlich stabileren Nachweis erbringen und die Theorie stützen.

Elemente des Graphen und Eigenschaften des Systems Aufbauend auf einer empirischen Ermittlung von Realwerten kann auch konkret die Darstellung verbessert werden. Durch eine Erweiterung im Graphen mit neuen (relevanten) Systemeigenschaften würden eventuelle neue Erkenntnisse sichtbar. Durch eine verstärkte Ausarbeitung am „Grundgerüst“ des Nachweises, dem Graphen, ist es möglich, neue Eigenschaften und vielleicht sogar zeitabhängige Verhaltensweisen des Systems formal zu repräsentieren, sodass diese neuen Werte später für ausgeprägtere Wert-Berechnungen nutzbar wären. Denkbar wären zum Beispiel folgende System-Eigenschaften: Komplexität der Elemente, Beziehungen zwischen Komponenten, Verteilung von Ressourcen, Auslastung und Relevanz von Elementen, Interaktionspunkte und -Möglichkeiten von Komponenten, verfeinerte Typen von *unmanaged* Datenredundanz, Einbeziehung der redundanten Bereiche innerhalb der Datenbestände, vererbte Redundanz über mehrere Generationen hinweg und anderes mehr. Ebenfalls könnten Spezialfälle des Graphen ausgearbeitet werden, wie

zum Beispiel der Fall, dass ein Datenbestand von einem anderen gleichzeitig *managed* und *unmanaged* redundant kopiert, was im bisherigen Graph-Modell nicht vorgesehen ist. Besonders interessant sind Eigenschaften, welche auch mathematisch interessant wären, wie zum Beispiel Zyklen-Berechnung und Invarianten-Auswertung. Allein die Ausweitung der angewendeten Methoden in einem *labeled, directed, annotated Multigraph* würde eine Menge neuer Erkenntnisse hervorbringen, da die Möglichkeiten dieser Art von Graphen in dieser Arbeit nur marginal ausgeschöpft wurden. All diese Eigenschaften können dabei helfen, die Kausalität zu stützen und die Quantifizierung zu verfeinern. Besonders kann man dabei die Möglichkeiten der Graph-Transformation hervorheben, welche es erlaubt, formale Regeln für Änderungen am Graphen durchzuführen. Dies kann sehr hilfreich sein, um eine Veränderung des Systems aus dem Zustand n zu $n + 1$ auf Grundlage des Graphen zu formalisieren. Dies würde eine sehr geeignete Methode für eine Simulation und Berechnung einer Systemevolution darstellen.

Ausarbeitung der Graph- und Wert-Metriken Der zweite Part, welcher neben der Darstellung konkret erweitert werden kann, ist die Messbarkeit auf selbiger. Die Erweiterung der Graph-Metriken wurde bereits kurz angeschnitten, aber auch die Wert-Metriken kommen in Betracht, wenn man neue Ansatzpunkte für zukünftige Arbeiten finden möchte. Beide Aspekte des Nachweises können sicherlich als Kernpunkt der Arbeit erweitert werden. Dabei würde eine „Ausarbeitung“ der Metriken in zwei verschiedene Richtungen gehen können: zum einen können bestehende Metriken verfeinert werden, zum anderen können sie erweitert werden. Beides sind relevante Punkte in der Bestimmung von Systemwerten. Dabei wird nicht unbedingt eine Erweiterung des Graphen vorausgesetzt, wie im vorherigen Abschnitt beschrieben. Die Metriken, wie sie jetzt sind, könnten einer Feinabstimmung unterzogen werden, in der jeder einzelne Faktor noch einmal genauer untersucht, seine Relevanz geprüft und mit der Realität abgeglichen wird. Dabei könnten auch neue Faktoren einbezogen werden, welche jetzt eine eher untergeordnete Rolle spielen oder gar nicht auftauchen, zum Beispiel die „Change-Rate“ von Datenbeständen, die „inhaltliche Nähe“ von Elementen oder die Schwierigkeit beim Auffinden der Datenredundanz. Aber auch bereits eingearbeitete Metriken können Veränderungen erfahren, zum Beispiel die Art und Weise, wie in *Corrective Maintenance* der strukturelle Ausgleich zwischen einzelnen Komponenten geschaffen wird. Eine Verbesserung der Metriken würde zu einer stabileren Kausalität (im naturwissenschaftlichen Sinne) und einer genaueren Quantifizierung des Wertverlusts führen. Dies könnte soweit führen, dass man die Metriken in Projektkosten-Berechnungen einfließen lässt.

Erweiterung des Programms zur System Simulation Dieser Punkt ist eher nebensächlich für die Theorie, dafür aber von Bedeutung beim Nachweis der Theorie. Immerhin wird mit dem „Simulation Program“ aufgezeigt, was genau im System vor sich geht und wie sich Metriken entwickeln, wenn die Evolution in nicht mehr manuell handhabbaren Schritten vonstatten geht. Dabei liegt besonderes Augenmerk nicht nur bei der tatsächlichen zufälligen Simulation einer Systemevolution, sondern auch in der Verbesserung „klassischer“ Werte wie zum Beispiel *Usability*, *Performance*, *Adaptivity* oder auch *Maintainability*. Zudem wären weitere Features durchaus machbar, wie eine entsprechende Umsetzung für

- automatisches Einlesen und Ausschreiben von Systemen in jedem Zustand,
- eine grafische Darstellung von den einzelnen Evolutionsschritten sowie deren Systemzuständen,
- verbesserte technische Umsetzung, Skalierbarkeit und Thread-Unterstützung,

- verbessertes Benutzerinterface mit intuitiver Bedienung,
- neue und verbesserte Funktionen in Bezug auf das System, zum Beispiel dem schnellen Einfügen von mehreren definierten Kanten oder einer zufälligen Erweiterung nur aus neuen Datenbeständen,
- eine Möglichkeit, die Metriken und den Graphen direkt im Programm anzupassen, oder
- technische Verbesserungen wie Graph-Datenbanken (statt Listen) für Elemente.

Dabei sind einige Punkte einfacher umzusetzen als andere. Allerdings wäre ein gut ausgereiftes Simulationstool ein großer Mehrwert für die hier dargestellten Folgerungen im Nachweis sowie darauf aufbauende weiterführende Arbeiten.

Ausarbeitung der Berechnung für den Systemwert sowie TtM und DevC Tatsächlich kann man auch das Resultat der Arbeit, den Systemwert k, noch erweitern. Bislang ist der Systemwert definiert durch den normalen Aufwand plus zusätzlich notwendige Investition *ARI*. Das Ergebnis äußert sich in einer Erhöhung in *TtM* und *DevC*. Aus beiden Faktoren wird der *Agility Value* gebildet, welcher, da *TtM* und *DevC* direkt aus dem *ARI* berechnet werden, sich gleich dem Systemwert verhält. Allerdings ist denkbar, dass nicht nur der *ARI* in eine erhöhte *TtM* einfließt, sondern auch die dadurch erhöhte *TtM* sich in Kosten für den Entwickler als *DevC* äußert. Bei beiden Berechnungen wurde von einer konstanten Umrechnung ausgegangen, was jedoch in der Praxis nicht direkt übertragbar ist, da in der Realität noch andere Faktoren einspielen, die in der Berechnung von *TtM* und *DevC* dieser Arbeit nicht betrachtet wurden, wie zum Beispiel Plattform-abhängige Entwicklungskosten oder Entwicklungskosten für Datenbestände (welche bislang in die Komponenten eingeflossen sind). Neben der Ermittlung für tatsächliche Werte für *TtM* und *DevC* gilt es auch noch, den Unterschied zu dem *Agility Value* herzustellen. Bislang unterscheiden sich k und *Agility Value* nur durch ihre Höhe zwischen den Werten 0 und 1. Aber es wäre möglich, dass sich beide unterschiedlich verhalten können, denn die Intention des *Agility Value* ist es, darzustellen, inwieweit sich das System gegen Änderungen „widersetzt“ (durch hohe Änderungskosten pro Änderung). Dagegen soll der Systemwert k darstellen, welchen Architektur-Wert (oder Wertverlust) das System besitzt. Da in dieser Arbeit der *Agility Value* und k direkt aus dem *ARI* abgeleitet werden, gilt es herauszufinden, inwieweit man beides voneinander unterscheiden kann, damit beide Werte ihrer Intention dienen.

Erfassung neuer, abgeleiteter Themenbereiche Ein sehr großes Themenfeld für die Entstehung neuer Erkenntnisse und Arbeiten ist die Ergründung weiterer Forschungsfelder, die in „Nachbarschaft“ mit dieser Arbeit stehen. Welche Punkte machen ein gutes System aus? Wie repräsentiert man Architektur in einem System? Wie kann man andere Architektur-Prinzipien darstellen? Und wie kann man sie als Systemwert k darstellen? Welche Möglichkeiten ergeben sich auch anderen mathematischen Konstrukten außer dem *labeled, directed, annotated Multigraph* und wie können sie eingebracht werden? Wie lassen sich reale Systeme in *ARI* auffassen? Wie lässt sich Redundanz und andere, ähnliche Phänomene der *Technical Debt* formal darstellen? Wie könnte ein Werkzeug für dessen Berechnung aussehen, welches nicht nur auf reiner Code-Basis sondern durch Architektur den Schaden einschätzt? Welche anderen Wege gibt es, Architektur einen fassbaren Wert zu verleihen?

All diese Fragen sind offene Themenbereiche, zu denen bisher nur wenig oder gar nichts in der Literatur zu finden ist. Dennoch wären die Beantwortung eine Bereicherung für diejenigen Personen an einem Projekt, welche sich mit Software-Qualität, -Design und -Architektur

beschäftigen, sei es für die Einschätzung des eigenen Systems, zur Überzeugung anderer Stakeholder oder einfach der Sicherheit halber, das Richtige zu tun.

7.3 Zusammenfassung und Einschätzung

Hier endet nun diese Arbeit in all ihren Facetten. Ausführlich wurden einzelne Schritte beschrieben, von der Motivation zu einem naturwissenschaftlichen Beweis, über die Architektur in einem System, dem Phänomen der Redundanz und der *unmanaged* Redundanz, der Methodik des Nachweises über Graphen, hin zu dem Nachweis selbst und damit der Bestätigung für einen kausalen Zusammenhang zwischen Architekturprinzipien und Wert von Software. Es wurden viele Methoden und Schritte zu Hilfe gezogen, um den Beweis zu stützen, welcher die Theorie bestätigen soll. Viele Annahmen wurden ausgewiesen und Faktoren als Begründung für quantitative Einschätzungen herangezogen. Neue Vorgehensweisen sind während des Verlaufes des Nachweises entstanden, wie die formale Darstellung von Architekturprinzipien, neue Darstellungsweisen der *Technical Debt* oder einer Auswertung des Systemwertes oder einer Einschätzung der Systemevolution.

Beginnend mit Kapitel 1 „Einleitung“ wurde das initiale Problem beschrieben, was das Ziel ist und warum diese Arbeit relevant ist. Im Kapitel 2 „Grundlagen von Softwaresystemen“ wurde genauer auf die Situation von großen Softwaresystemen eingegangen, und beschrieben, wie Softwaresysteme als Investitionsgut anzusehen sind, was den „Wert“ dieser Systeme ausmacht, wie sich in diesen die Architektur bestimmt. Weiterhin wurden Architekturprinzipien genannt und die Auswirkungen positiver und negativer Architektur aufgezeigt. Das Ende dieses Kapitels bildete eine Exkursion in langlebige Software und wie sie sich definiert. Mit Kapitel 3 „Methodik und Metriken“ begann dann die Eigenforschung und die Vorgehensweisen, Methoden und Bezüge für den kommenden Nachweis wurden vorgestellt. Dabei wurde das erste große Konzept von „Darstellung und Messbarkeit“ erläutert, und wie sich dadurch der Nachweis definieren wird. Ebenfalls sind in diesem Kapitel die Graphen beschrieben worden, welche als Darstellung für den Nachweis dienten. Am Ende des Kapitels wurden Metriken vorgestellt, welche den Kern des Nachweises bilden sollten. Während im folgenden Kapitel 4 „Redundanz“ noch einmal das Thema der Redundanz, Datenredundanz sowie der Unterschied zwischen *managed* und *unmanaged* dargestellt wurden, begann der eigentliche Nachweis in Kapitel 5 „Der Nachweis“. Schritt für Schritt wurden die neuen Metriken erstellt und genutzt um den Nachweis aufzubauen, über die Definition des Graphen, dessen Metriken hin zu dem Wert des Systems durch die Wert-Metriken und dem dadurch definierten Systemwert k. Am Ende des Kapitels stand der Nachweis in Form von verschiedenen Metriken, welche zum einen den direkten Zusammenhang zwischen Architekturprinzipien-Treue des Systems (dem Graphen) und dem monetären Wert (als *TtM* und *DevC*) darstellten, zum anderen die Möglichkeit gaben, eine erste geeignete quantitative Summe als „Systemwert“ dafür zu bestimmen, was passiert, wenn diese Architekturprinzipien vernachlässigt werden. Zum Schluss wurde in Kapitel 6 „System Simulation“ ein Tool programmiert, welches Aussage darüber lieferte, wie sich die im vorherigen Kapitel erstellten Metriken in großen und aufwendigen Systemen und deren Evolution auswirken. Generell wurden dabei keine Defizite festgestellt, jedoch konnte festgestellt werden, dass die Quantifizierung durch die Metriken durchaus noch verfeinert und verbessert werden kann. Ansonsten eignet sich das Tool zudem sehr gut, mit diversen Parametern und Metriken zu experimentieren, um in eventuellen zukünftigen Projekten den kausalen Zusammenhang zu stärken oder die Metriken selbst zu verbessern.

Gemäß Schuster *et al.* [48] sollten am Anfang jeder Forschungsarbeit zwei Dinge klargestellt

werden: was ist der eigene Beitrag und warum ist dieser so wichtig? Beides wird wie folgt zusammengefasst:

Haupt-Erkenntnisse

1. Softwarearchitektur hat ihren eigenen, von betriebswirtschaftlichen Sichtweisen unabhängigen Wert für das Investitionsgut „Software“.
2. Architekturprinzipien-Treue dient als Basis für gute Architektur und zugehörigen gesteigerten Wert der Software bezüglich der Evolution, was formal und kausal aufgezeigt werden kann.
3. Architekturprinzipien können quantitativ (monetär) als Wert für Softwarearchitektur angegeben werden.

Wo nun die Haupt-Erkenntnisse beleuchtet wurden, steht es noch offen, deren Wichtigkeit für entsprechende Thematiken zu präsentieren. Warum eine derart lange theoretische Arbeit zu den genannten Erkenntnissen? Die Sinnhaftigkeit von naturwissenschaftlichen Beweisen wurde schon mehrmals angesprochen, wie sieht es aber im konkreten Fall dieser Arbeit aus? Zuerst einmal sei gesagt, dass diese Arbeit eine gewisse Sicherheit mitbringt. Die hier vorgestellte Theorie mag der ausschlaggebende Punkt am Verhandlungstisch sein, warum in einem Projekt mehr Zeit und Geld für gute Architektur genehmigt wird, da der Verfall des Systems durch vernachlässigte Architekturprinzipien nun erfassbar geworden ist. Selbst wenn sich die hier angenommenen Werte nicht realistisch verhalten wäre dies kein Problem. Dann steht es jedem frei, seine Anpassungen vorzunehmen, sodass diese wieder belastbar werden. Die Vorgehensweise wurde aufgezeigt, von der initialen Erfassung der Prinzipien, über deren formale Darstellung, deren Messung bis schlussendlich deren Auswertung. Wichtig ist, dass ein Weg bereitet wurde, um die Annahme, dass sich gute Architektur auszahlt, auch tatsächlich formal darzulegen. Und dieser Weg ist die Kernessenz aus der vorliegenden Arbeit, viel wichtiger als die Details wie konkrete Belegung von Faktoren oder formale Beschreibung von Systemelementen.

A Appendix

A.1 Berechnung der Metriken am Beispielsystem

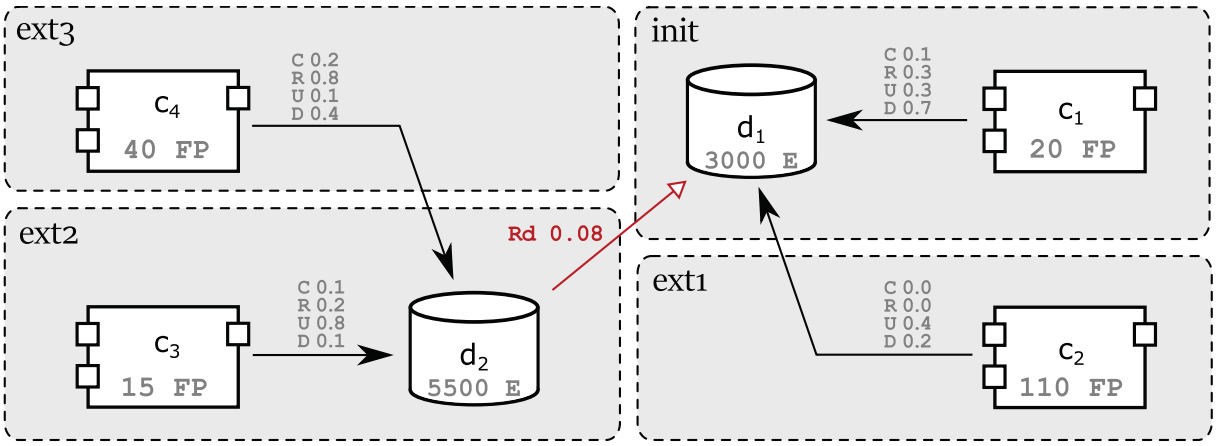


Abbildung A.1: Das System als Berechnungsgrundlage

Die Berechnung von Systemen per Metriken kann sehr aufwendig werden. Dies soll an dem referenzierten System vorgenommen werden, welches noch einmal in Abbildung A.1 aufgezeigt ist. Insgesamt werden alle drei Wert-Metriken ausgeführt, welche jeweils über entweder alle Komponenten oder alle Datenbestände iterieren. Man befindet sich im Systemzustand $\mathbb{E}_3^+$, was unter anderem für das $r(\lambda)$ der *Corrective Maintenance* wichtig ist. Man beachte, dass alle Elemente nur einfach erstellt und bislang noch *nicht* erweitert wurden, sodass $cr(c_i) = FP(c_i)$ ist. Die Rechnungen werden dahingehend vereinfacht, dass nur über relevante Elemente iteriert wird, sodass etwaige Berechnungen wie $\varphi_{c_2 d_2}$ ausgelassen werden, da keine Verbindung von c_2 nach d_2 existiert und dieser Iterationsschritt am Ende 0 addieren würde. Das Ergebnis des Simulationsprogramms per `analyse()` ist Folgendes und wird anschließend händisch nachgerechnet:

```
***** Summary *****
ARI:    2.0629957 (0.432 + 0.010354337 + 1.6206412) * q
k:      0.9889717
AV:     0.61129063 (std: 0.625)
TtM+:   4.13 days
DevC+: 1652.0 euros
Your system (185.0 FP, 187.06299 eFP) in extension 3 contains:
        #C = 4, #DB = 2, #Use = 4, #Red = 1
```

Quellcode A.1: Das vorausberechnete Ergebnis von `analyse()`

Hilfswerte

$$
\begin{aligned}
\varphi_{c_1 d_1} \cdot \varphi' &= 0.1 \cdot 0.25 + 0.3 \cdot 0.1 + 0.3 \cdot 0.4 + 0.7 \cdot 0.25 && = 0.35 \\
\varphi_{c_2 d_1} \cdot \varphi' &= 0.0 \cdot 0.25 + 0.0 \cdot 0.1 + 0.4 \cdot 0.4 + 0.2 \cdot 0.25 && = 0.21 \\
\varphi_{c_3 d_2} \cdot \varphi' &= 0.1 \cdot 0.25 + 0.2 \cdot 0.1 + 0.8 \cdot 0.4 + 0.1 \cdot 0.25 && = 0.39 \\
\varphi_{c_4 d_2} \cdot \varphi' &= 0.2 \cdot 0.25 + 0.8 \cdot 0.1 + 0.1 \cdot 0.4 + 0.4 \cdot 0.25 && = 0.27 \\
N(d_1) &= (FP_{c_1} \cdot \varphi_{c_1 d_1} \cdot \varphi') + (FP_{c_2} \cdot \varphi_{c_2 d_1} \cdot \varphi') \\
&= (20 \cdot 0.35) + (110 \cdot 0.21) && = 30.10 \\
N(d_2) &= (FP_{c_3} \cdot \varphi_{c_3 d_2} \cdot \varphi') + (FP_{c_4} \cdot \varphi_{c_4 d_2} \cdot \varphi') \\
&= (15 \cdot 0.39) + (40 \cdot 0.27) && = 16.65 \\
\tilde{E} &= Median([3000, 5500]) \Rightarrow \frac{1}{2}(3000 + 5500) && = 4250
\end{aligned}
$$

Adaptive Maintenance

$$
\begin{aligned}
c_1 &= 20 \cdot (\delta_{c_1 d_1} \cdot \varphi_{c_1 d_1} \cdot \varphi' \cdot (0)) \cdot q_a \\
&= 20 \cdot (0 \cdot 0.35 \cdot (0)) \cdot q_a && = 0 \\
c_2 &= 110 \cdot (\delta_{c_2 d_1} \cdot \varphi_{c_2 d_1} \cdot \varphi' \cdot (0)) \cdot q_a \\
&= 110 \cdot (1 \cdot 0.21 \cdot (0)) \cdot q_a && = 0 \\
c_3 &= 15 \cdot (\delta_{c_3 d_2} \cdot \varphi_{c_3 d_2} \cdot \varphi' \cdot (0.08)) \cdot q_a \\
&= 15 \cdot (0 \cdot 0.39 \cdot (0.08)) \cdot q_a && = 0 \\
c_4 &= 40 \cdot (\delta_{c_4 d_2} \cdot \varphi_{c_4 d_2} \cdot \varphi' \cdot (0.08)) \cdot q_a \\
&= 40 \cdot (1 \cdot 0.27 \cdot (0.08)) \cdot q_a && = 0.864
\end{aligned}
$$

$$
\begin{aligned}
y_a &= 0.864 \\
y_a \cdot q_a &= y_a \cdot 0.5 = 0.432
\end{aligned}
$$

Corrective Maintenance

$$
\begin{aligned}
c_1 &= 20 \cdot \left(\frac{3-0}{3}\right) \cdot \left(\frac{N(d_1) \cdot N(d_2)}{(20+110+15+40)^2} \cdot \sqrt{0}\right) \\
&= 20 \cdot 1 \cdot \left(\frac{30.1 \cdot 16.65}{34225} \cdot 0\right) && = 0 \\
c_2 &= 110 \cdot \left(\frac{3-1}{3}\right) \cdot \left(\frac{N(d_1) \cdot N(d_2)}{(20+110+15+40)^2} \cdot \sqrt{0}\right) \\
&= 110 \cdot \frac{2}{3} \cdot \left(\frac{30.1 \cdot 16.65}{34225} \cdot 0\right) && = 0 \\
c_3 &= 15 \cdot \left(\frac{3-2}{3}\right) \cdot \left(\frac{N(d_1) \cdot N(d_2)}{(20+110+15+40)^2} \cdot \sqrt{0.08}\right) \\
&= 15 \cdot \frac{1}{3} \cdot \left(\frac{30.1 \cdot 16.65}{34225} \cdot 0.28284\right) && \approx 0.0207 \\
c_4 &= 40 \cdot \left(\frac{3-3}{3}\right) \cdot \left(\frac{N(d_1) \cdot N(d_2)}{(20+110+15+40)^2} \cdot \sqrt{0.08}\right) \\
&= 40 \cdot 0 \cdot \left(\frac{30.1 \cdot 16.65}{34225} \cdot 0.28284\right) && = 0
\end{aligned}
$$

$$
\begin{aligned}
y_c &= 0.0207 \\
y_c \cdot q_c &= y_c \cdot 0.5 = 0.01035
\end{aligned}
$$

Preventive Maintenance

$$
\begin{aligned}
d_1 &= \left(\frac{3000}{\tilde{E}} \cdot N(d_1)^0\right) \Rightarrow 0, \text{ da } [\overline{Rd}_{d_1 d_2} = 0] && = 0 \\
d_2 &= \left(\frac{5500}{\tilde{E}} \cdot N(d_2)^{0.08}\right) \\
&= \left(\frac{5500}{4250} \cdot 16.65^{0.08}\right) && \approx 1.6206
\end{aligned}
$$

$$
\begin{aligned}
y_p &= 1.6206 \\
y_p \cdot q_p &= y_p \cdot 1.0 = 1.6206
\end{aligned}
$$

Systemwert

$$\begin{aligned} ARI/y &= y_a + y_c + y_c \\ &= 0.432 + 0.01035 + 1.6206 \\ &\underline{\underline{= 2.06295}} \end{aligned}$$

$$\begin{aligned} k &= \frac{20 + 110 + 15 + 40}{(20 + 110 + 15 + 40) + y} \\ &= \frac{185}{185 + 2.06295} \\ &\underline{\underline{\approx 0.98897}} \end{aligned}$$

$$\begin{aligned} TtM+ &= y \cdot 2 \\ &\underline{\underline{\approx 4.13 \text{ Tage}}} \end{aligned}$$

$$\begin{aligned} DevC+ &= 50 \ \frac{€}{\text{h}} \cdot 8 \text{ h} \cdot 4.13 \text{ Tage} \\ &\underline{\underline{\approx 1652 €}} \end{aligned}$$

$$\begin{aligned} AV &= \frac{185^2}{\left(TtM \cdot \frac{DevC}{1000}\right)} \\ &= \frac{185^2}{((185 \cdot 2) + 4.13) \cdot \left(\frac{(185 \cdot 2 \cdot 8 \cdot 50) + 1652}{1000}\right)} \\ &\underline{\underline{\approx 0.61128}} \end{aligned}$$

A.2 Belegung der Parameter des Programms für das Testszenario

Hinweis: Diese Parameter sind durch Austesten entstanden, und *nicht* empirisch aus real existierenden Systemen ermittelt. Dennoch stellen diese Werte mindestens ein halbwegs realistisches Szenario dar. Mehr dazu in zugehörigen Kapitel 6.3.1 „Bezug zur Simulation“.

general options	How many days per FP	2
	How many euros per hour	50
	Which random seed	1337
	Value of global 'q_a'	0.5
	Value of global 'q_c'	0.5
	Value of global 'q_p'	1.0
new item/use/redundancy settings	New items (means 'system changes') [mean]	5.0
	New items (means 'system changes') [deviation]	3.0
	New use-relations [mean]	1.0
	New use-relations [deviation]	1.2
	New redundancies [mean]	0.2
	New redundancies [deviation]	0.3
	Relative amount of new components	0.7
	Relative amount of new databases	0.3
	Relative amount of adding new components	0.1
	Relative amount of extending old components	0.9
	Relative amount of adding new databases	0.1
	Relative amount of extending old databases	0.9
	Relative amount of STD redundancies	0.7
	Relative amount of SYNC redundancies	0.3
	Factor of 'phi' in new use-relations	0.8
size settings	Size of new components [mean]	40
	Size of new components [deviation]	20
	Extending size of old components (percentage!) [mean]	1.5
	Extending size of old components (percentage!) [deviation]	2.5
	Size of new databases [mean]	20000
	Size of new databases [deviation]	10000
	Extending size of old databases (percentage!) [mean]	2.5
	Extending size of old databases (percentage!) [deviation]	1.5
redundancy	Min value of new STD redundancies	0.001
	Max value of new STD redundancies	0.15
	Min value of new SYNC redundancies	0.05
	Max value of new SYNC redundancies	0.35

Tabelle A.1: Parameter für das Testszenario

B Inhalt elektronischer Datenträger (DVD)

- Textdokumente der Arbeit im PDF-Format
- Quellcode und Artefakte des Simulationsprogramms
- Folien des Verteidigungsvortrages

Für eine detaillierte Inhaltsübersicht des Datenträgers wird auf das dort hinterlegte Inhaltsverzeichnis verwiesen.

Literaturverzeichnis

[1] Ahn, Yunsik, Jungseok Suh, Seungryeol Kim und Hyunsoo Kim: The software maintenance project effort estimation model based on function points. Journal of Software Maintenance and Evolution: Research and Practice, 15(2):71–85, 2003.

[2] Allen, Edward B., Taghi M. Khoshgoftaar und Ye Chen: Measuring coupling and cohesion of software modules: an information-theory approach. In: Seventh International Software Metrics Symposium, METRICS '01, Seiten 124–134. IEEE Press, 2001.

[3] Bass, Len, Paul Clements und Rick Kazman: Software Architecture in Practice. Addison-Wesley Professional, 1998.

[4] Brown, Nanette, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan und Nico Zazworka: Managing Technical Debt in Software-Reliant Systems. In: Proceedings of the FSE/SDP workshop on Future of software engineering research, Seiten 47–52, 2010.

[5] Buchholz, Scott und David Sisk: Technical debt reversal - Lowering the IT debt ceiling. Seiten 67–77. Deloitte Consulting LLP, 2014.

[6] Bungartz, Hans-Joachim, Stefan Zimmer, Martin Buchholz und Dirk Pflüger: Modellbildung und Simulation. Springer-Verlag Berlin Heidelberg, 2009.

[7] Carzaniga, Antonio, Andrea Mattavelli und Mauro Pezzè: Measuring Software Redundancy. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15, Seiten 156–166. IEEE Press, 2015.

[8] Chapin, Ned, Joanne E. Hale, Khaled Md. Khan, Juan F. Ramil und Wui-Gee Tan: Types of software evolution and software maintenance. Journal of Software Maintenance and Evolution: Research and Practice, (13):3–30, 2001.

[9] Charette, Robert N: This car runs on code. IEEE spectrum, 46(3):3, 2009.

[10] Chlistalla, Michael: High-frequency trading. Deutsche Bank Research, 2011.

[11] Cunningham, Ward: The WyCash Portfolio Management System. In: Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum), OOPSLA '92, Seiten 29–30, 1992.

[12] Curtis, Bill, Jay Sappidi und Alexandra Szynkarski: Estimating the Size, Cost, and Type of Technical Debt. In: MTD '12 Proceedings of the Third International Workshop on Managing Technical Debt, Seiten 49 – 53, 2012.

[13] Diestel, Reinhard: Graphentheorie. Springer-Verlag Berlin Heidelberg, 2010.

[14] Dimitropoulos, Xenofontas, Dmitri Krioukov, Amin Vahdat und George Riley: Graph annotations in modeling complex network topologies. In: ACM Transactions on Modeling and Computer Simulation, 2009.

[15] Duden.de: Redundanz. http://www.duden.de/rechtschreibung/Redundanz, 2016. [Online, Mai 2016].

[16] Fenton, Norman E. und James Bieman: Software Metrics - A Rigorous and Practical Approach. CRC Press, 2015.

[17] Fenton, Norman E. und Martin Neil: Software Metrics: Roadmap. In: Proceedings of the Conference on The Future of Software Engineering, ICSE '00, Seiten 357–370, 2000.

[18] Fowler, Martin: Technical Debt Quadrant. http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html, 2009. [Online, November 2015].

[19] Furrer, Frank J.: Future-proof Software Systems (Vorlesungsreihe, TU Dresden, Fakultät Informatik). http://tu-dresden.de/die_tu_dresden/fakultaeten/fakultaet_informatik/smt/st/studium?subject=304&lang=en&leaf=1&head=2, 2015. [Online, November 2015].

[20] Furrer, Frank J.: Zukunftsfähige Softwaresysteme. Informatik-Spektrum, 39(3):194–202, Jun 2016.

[21] Gierow, Hauke (Golem.de): Mozilla schmeißt SHA-1 raus - und gleich wieder rein. http://www.golem.de/news/firefox-mozilla-schmeisst-sha-1-raus-und-gleich-wieder-rein-1601-118438.html, 2016. [Online, Januar 2016].

[22] Gorton, Ian: Essential Software Architecture. Springer-Verlag Berlin Heidelberg, 2011.

[23] Greefhorst, Danny und Erik Proper: Architecture Principles. Springer-Verlag Berlin Heidelberg, 2011.

[24] Harrison, Warren und Curtis Cook: Insights on improving the maintenance process through software measurement. In: Conference on Software Maintenance, 1990, Proceedings, Seiten 37–45. IEEE, 1990.

[25] Held, Andrea: Oracle 10g Hochverfügbarkeit. Addison-Wesley Verlag, 2004.

[26] IEEE: Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Board, 2000.

[27] Kastens, Uwe und Hans Kleine Büning: Modellierung: Grundlagen und formale Methoden. Carl Hanser Verlag, 2014.

[28] Khorikov, Vladimir: Cohesion and Coupling: the difference. http://enterprisecraftsmanship.com/2015/09/02/cohesion-coupling-difference/, 2015. [Online, Februar 2016].

[29] Klinger, Tim, Peri Tarr, Patrick Wagstrom und Clay Williams: An Enterprise Perspective on Technical Debt. In: Proceedings of the 2nd Workshop on Managing Technical Debt, MTD '11, Seiten 35–38, 2011.

[30] Kroll, Joshua A., Ian C. Davey und Edward W. Felten: The Economics of Bitcoin Mining, or Bitcoin in the Presence of Adversaries. The Twelfth Workshop on the Economics of Information Security (WEIS 2013), 2013.

[31] Krumke, Sven O. und Hartmut Noltemeier: Graphentheoretische Konzepte und Algorithmen. Teubner, 2005.

[32] Letouzey, Jean-Louis: The SQALE Method for Evaluating Technical Debt. In: MTD '12 Proceedings of the Third International Workshop on Managing Technical Debt, Seiten 31– 36, 2012.

[33] Lientz, B. P., E. B. Swanson und G. E. Tompkins: Characteristics of Application Software Maintenance. Communications of the ACM, 21(6):466–471, 1978.

[34] Lilienthal, Carola: Langlebige Software-Architekturen. dpunkt.verlag, 2016.

[35] Lorenz, Mark und Jeff Kidd: Object-oriented software metrics: a practical guide. Prentice-Hall, Inc, 1994.

[36] Mahadevan, Priya, Dmitri Krioukov, Kevin Fall und Amin Vahdat: Systematic topology analysis and generation using degree correlations. In: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications, Seiten 135–146, 2006.

[37] Martin, Robert C.: Design Principles and Design Patterns. 2000. `https://scm0329.googlecode.com/svn-history/r78/trunk/book/Principles_and_Patterns.pdf` [Online, März 2016].

[38] Matson, Jack E., Bruce E. Barrett und Joseph M. Mellichamp: Software Development Cost Estimation Using Function Points. Transactions on Software Engineering, 20(4), 1994.

[39] McCandless, David: Million Lines of Code - Information is Beautiful. `http://www.informationisbeautiful.net/visualizations/million-lines-of-code/`, 2016. [Online, Juni 2016].

[40] Mey, Stefan (Heise.de): Bitcoin-Technik: Blockchain als Wunderwaffe der Finanzwirtschaft. `http://www.heise.de/newsticker/meldung/Bitcoin-Technik-Blockchain-als-Wunderwaffe-der-Finanzwirtschaft-2699919.html`, 2015. [Online, Juni 2016].

[41] Murer, Stephan, Bruno Bonati und Frank J. Furrer: Managed Evolution - A Strategy for Very Large Information Systems. Springer-Verlag Berlin Heidelberg, 2011.

[42] Nugroho, Ariadi, Joost Visser und Tobias Kuipers: An Empirical Model of Technical Debt and Interest. In: Proceedings of the 2nd Workshop on Managing Technical Debt, MTD '11, Seiten 1–8, 2011.

[43] Oquendo, Flavio, Brian Warboys und Ron Morrison (Herausgeber): Software Architecture, First European Workshop EWSA 2014. Springer, 2004.

[44] Owens, Thomas (Stackexchange.com): How can I quantify the amount of technical debt that exists in a project? http://programmers.stackexchange.com/questions/135993/how-can-i-quantify-the-amount-of-technical-debt-that-exists-in-a-project, 2012. [Online, November 2015].

[45] Rötzer, Florian (Heise.de): Heiß umkämpft: Programme zum High-Frequency-Trading. http://www.heise.de/newsticker/meldung/Heiss-umkaempft-Programme-zum-High-Frequency-Trading-752721.html, 2009. [Online, Juni 2016].

[46] Sadalage, Pramod J. und Martin Fowler: NoSQL Distilled. Addison-Wesley Professional, 2013.

[47] Scherschel, Fabian A. (Heise.de): Totgesagte leben länger: Facebook und Cloudflare setzen weiter auf SHA-1. http://www.heise.de/newsticker/meldung/Totgesagte-leben-laenger-Facebook-und-Cloudflare-setzen-weiter-auf-SHA-1-3041665.html, 2015. [Online, Dezember 2015].

[48] Schuster, Ethel, Haim Levkowitz und Osvaldo N. Oliveira: Writing scientific papers in english successfully. hyprtek.com, inc., 2014.

[49] Singh, Simon: Fermats letzter Satz. Deutscher Taschenbuch Verlag, 2011.

[50] Söderberg, Bo: Random graphs with hidden color. Phys. Rev. E, 68:015102, Jul 2003.

[51] Völkl, Christina: Innenwinkelsumme des Dreiecks. https://www2.uni-wuerzburg.de/dmuw-vhb/demo/geo_cd_nbg/dreiecke/beweise/drset_iwisu.html, 2013. [Online, Juni 2016].

[52] Waser, André: Die logarithmische Verteilung in der Natur. http://www.andre-waser.ch/Publications/DieLogarithmischeVerteilungInDerNatur.pdf, 2003.

[53] Weber, Jens H., Anthony Cleve, Loup Meurice und Francisco J. B. Ruiz: Managing Technical Debt in Database Schemas of Critical Software. In: 6th IEEE International Workshop on Managing Technical Debt, Seiten 43 – 46. IEEE, 2014.

[54] Wikipedia: Aufwandsschätzung (Softwaretechnik). https://de.wikipedia.org/w/index.php?title=Aufwandssch%C3%A4tzung_(Softwaretechnik)&oldid=146790248, 2015. [Online, Juni 2016].

[55] Wikipedia: Graphentheorie. https://de.wikipedia.org/w/index.php?title=Graphentheorie&oldid=154433123, 2015. [Online, Februar 2016].

[56] Wikipedia: Kronecker-Delta. https://de.wikipedia.org/w/index.php?title=Kronecker-Delta&oldid=146234809, 2015. [Online, Februar 2016].

[57] Wikipedia: Reductio ad absurdum. https://de.wikipedia.org/w/index.php?title=Reductio_ad_absurdum&oldid=148902043, 2015. [Online, Februar 2016].

[58] Wikipedia: Technische Schulden. https://de.wikipedia.org/w/index.php?title=Technische_Schulden&oldid=147523980, 2015. [Online, Juni 2016].

[59] Wikipedia: Time-to-Market. https://de.wikipedia.org/w/index.php?title=Time-to-Market&oldid=149618860, 2015. [Online, Juni 2016].

[60] Wikipedia: Adjazenzmatrix. https://de.wikipedia.org/w/index.php?title=Adjazenzmatrix&oldid=154299660, 2016. [Online, März 2016].

[61] Wikipedia: Beweis (Logik). https://de.wikipedia.org/w/index.php?title=Beweis_(Logik)&oldid=150748945, 2016. [Online, Juni 2016].

[62] Wikipedia: Beweis (Mathematik). https://de.wikipedia.org/w/index.php?title=Beweis_(Mathematik)&oldid=142717046, 2016. [Online, Juni 2016].

[63] Wikipedia: Median. https://de.wikipedia.org/w/index.php?title=Median&oldid=155397366, 2016. [Online; Stand 23. Juni 2016].

[64] Wikipedia: Normalverteilung. https://de.wikipedia.org/w/index.php?title=Normalverteilung&oldid=154666133, 2016. [Online, März 2016].

[65] Wikipedia: Satz von Euklid. https://de.wikipedia.org/w/index.php?title=Satz_von_Euklid&oldid=150480516, 2016. [Online, Mai 2016].

[66] Woods, Viveca und Rob van der Meulen: Gartner Says Worldwide IT Spending is Forecast to Grow 0.6 Percent in 2016. http://www.gartner.com/newsroom/id/3186517, 2016. [Online, Juni 2016].

[67] Wunderlich-Pfeiffer, Frank (Golem.de): In den Neunzigern stürzte alles ab. http://www.golem.de/news/softwarefehler-in-der-raumfahrt-in-den-neunzigern-stuerzte-alles-ab-1511-117537.html, 2015. [Online, November 2015].

[68] Zegura, Ellen W., Kenneth L. Calvert und Michael J. Donahoo: A Quantitative Comparison of Graph-based Models for Internet Topology. IEEE/ACM Trans. Netw., 5(6):770–783, Dezember 1997.

Abbildungsverzeichnis

2.1 Der Wert einer Software. 16
2.2 Auswirkungen vernachlässigter Architektur bei Software 20
2.3 Der Einfluss auf ein Projekt 22
2.4 Die Balance zwischen *Business Requirements* und *Architecture Principles* 23
2.5 Das Koordinatensystem von *Agility*, *Resilience* und *Business Value* 30
2.6 Die Zusammenhänge bei der Entwicklung neuer, zukunftsfähiger Software 32
2.7 Die Kernidee der *Managed Evolution*: der *Evolution Channel* 34

3.1 Das Prinzip *Darstellung und Messbarkeit* vom Problemraum zur Lösung 36
3.2 Ein Beispielgraph für Netzwerke aus Dimitropoulos *et al.* [14] 40
3.3 Verschiedene Möglichkeiten eines Graphen 42

4.1 Die Einflüsse der verschiedenen Ursachen von Kosten 56

5.1 Das Vorgehen der Arbeit bezüglich Theorie, Hypothese und Nachweis 60
5.2 Die drei strukturellen Kosten/Wert-Bereiche 61
5.3 Ein zufälliges Beispiel eines *labeled, annotated, directed Multigraphen* 63
5.4 Die Knoten und Kanten des *labeled, annotated, directed Multigraphen* 67
5.5 Die Annotationen der Knoten und Kanten 69
5.6 Die Redundanz-Kanten und Erweiterungen im Graphen 71
5.7 Die Funktionsweise der vererbten Redundanz: nur B wird beachtet 73
5.8 Die Evolutionstheorie: vom `init`-System zu einem komplexen Softwaresystem . . 75
5.9 Die Logik hinter den Wert-Metriken 78
5.10 Die inhaltliche Größe von Komponenten und Datenbeständen 79
5.11 Ein Beispiel, in dem gesplittete Informationen als eine *Entity* zählen 81
5.12 Die Nutzung einer Komponente auf einem Datenbestand 82
5.13 Die Beziehung der Redundanz: Standard, vererbt und synchronisiert 85
5.14 Die Fremdheit eines Datenbestandes σ und σ' 89
5.15 Die Zusammensetzung der *Adaptive Maintenance* 100
5.16 Der Verlauf der gewählten Funktion für $r(\lambda)$ 103
5.17 Die Zusammensetzung der *Corrective Maintenance* 106
5.18 Die Zusammensetzung der *Preventive Maintenance* 110
5.19 Die letzten Schritte der kausalen Kette im Nachweis 112
5.20 Die Wert-Metriken 113
5.21 Das Verhalten der Metriken bei Anstieg der Redundanz $\overline{Rd}$ 117
5.22 Verlauf des Systemwertes k 122

6.1 *Use-Cases* im Simulation-Tool 131
6.2 Die Übertragung des Graph-Modells in die vierteilige Listenstruktur 134
6.3 Das Programm als UML-Diagramm 136
6.4 Das Programm als Graph dieser Arbeit 137

6.5 Die Ausgangssituation: das System als Input 141
6.6 Die Auswirkungen der *unmanaged* Datenredundanz im System (1) 146
6.7 Die Auswirkungen der *unmanaged* Datenredundanz im System (2) 148

7.1 Die einzelnen eingeführten Methodiken für den Nachweis 151

A.1 Das System als Berechnungsgrundlage i

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, den 27.06.2016